

Алгоритмы на Java

4-Е ИЗДАНИЕ

РОБЕРТ СЕДЖВИК | КЕВИН УЭЙН

Алгоритмы на Java

4-Е ИЗДАНИЕ

Algorithms

FOURTH EDITION

Robert Sedgewick
and
Kevin Wayne

Princeton University

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Алгоритмы на Java

4-Е ИЗДАНИЕ

Роберт Седжвик
Кевин Уэйн



Москва • Санкт-Петербург • Киев
2013

ББК 32.973.26-018.2.75
С28
УДК 681.3.07

Издательский дом "Вильямс"
Зав. редакцией *С.Н. Тригуб*
Перевод с английского *А.А.Моргунова*
Под редакцией *Ю.Н. Артеменко*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Седжвик, Роберт, **Уэйн**, Кевин.

С28 Алгоритмы на Java, 4-е изд. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2013. — 848 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1781-2 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание
Роберт Седжвик, Кевин Уэйн

Алгоритмы на Java, 4-е издание

Верстка *Т.Н. Артеменко*
Художественный редактор *В.Г. Павлютин*

Подписано в печать 31.07.2012. Формат 70×100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 68,37. Уч.-изд. л. 56,6.
Тираж 1500 экз. Заказ № 3239.

Первая Академическая типография "Наука"
199034, Санкт-Петербург, 9-я линия, 12/28

ООО "И. Д. Вильямс", 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-1781-2 (рус.)
ISBN 978-0-321-57351-3 (англ.)

Оглавление

Содержание

Посвящается Адаму, Эндрю, Бретту, Робби и особенно Линде

Посвящается Джеки и Алекс

Об авторах

Роберт Седжвик с 1985 г. является профессором в области вычислительной техники в Принстонском университете, где был основателем и заведующим кафедрой вычислительной техники. Он занимал должности внештатного научного сотрудника в Xerox PARC, Институте анализа в области обороны и INRIA, и является членом совета директоров в Adobe Systems. Направления исследовательской работы профессора Седжвика включают аналитическую комбинаторику, проектирование и анализ структур данных и алгоритмов, а также визуальные средства создания программ. Его книги, посвященные алгоритмам, за последние 30 лет претерпели множество изданий на разных языках. В соавторстве с Кевином Уэйном он написал широко известную книгу *Introduction to Programming in Java: An Interdisciplinary Approach* (Addison-Wesley, 2008 г.).

Кевин Уэйн — старший преподаватель кафедры вычислительной техники в Принстонском университете, в котором учился с 1998 г. Он получил степень доктора философии в области исследования операций и организации производства в Корнеллском университете. Его направления исследовательской работы включают проектирование, анализ и реализацию алгоритмов, особенно для графов и дискретной оптимизации. В соавторстве с Робертом Седжвиком он написал широко известную книгу *Introduction to Programming in Java: An Interdisciplinary Approach* (Addison-Wesley, 2008 г.).

Предисловие

Эта книга задумана как обзор наиболее важных на сегодняшний день компьютерных алгоритмов и как сборник фундаментальных приемов для все большего количества людей, которым они нужны. Она оформлена в виде учебника для второго курса изучения вычислительной техники, когда студенты уже владеют базовыми навыками программирования и знакомы с компьютерными системами. Книга может быть также полезна для самообразования или в качестве справочника тем, кто занят разработкой компьютерных систем или прикладных программ, поскольку она содержит реализации полезных алгоритмов и подробную информацию о характеристиках производительности и клиентских программах. Широкий охват материала делает данную книгу удобным введением в эту область.

Изучение алгоритмов и структур данных — основа любого компьютерного курса, не только для программистов и изучающих вычислительную технику. Каждый, кто пользуется компьютером, хочет, чтобы он работал быстрее и решал более крупные задачи. Алгоритмы в данной книге представляют собой сборник основных знаний, полученных за последние 50 лет и признанных незаменимыми. От моделирования системы N тел в физике до расшифровки генетического кода в молекулярной биологии — описанные здесь основные методы необходимы в любой области научных исследований; от систем архитектурного проектирования до моделирования самолетов они необходимы в инженерных расчетах; и от систем управления базами данных до механизмов поиска в Интернете они представляют собой необходимые части современных программных систем. Здесь приведено лишь несколько примеров: по мере расширения области применения компьютерных приложений растет и влияние описанных здесь базовых методов.

Прежде чем приступить к разработке фундаментального подхода к изучению алгоритмов, мы разработаем типы данных для стеков, очередей и других низкоуровневых абстракций, которые мы затем будем использовать на протяжении всей книги. Затем мы рассмотрим фундаментальные алгоритмы для сортировки, поиска, графов и строк. Последняя глава представляет собой обзор изложенного в книге материала в более широком контексте.

Отличительные черты

Назначение этой книги — изучение алгоритмов, которые наиболее широко применяются на практике. В книге изучается широкий спектр алгоритмов и структур данных и содержится объем информации о них, достаточный для уверенной реализации, отладки и работы в реальных приложениях в любой вычислительной среде. Этот подход включает в себя следующие аспекты.

- **Алгоритмы.** Наши описания алгоритмов основаны на полных реализациях и на анализе работы этих программ с помощью согласованного набора примеров. Вместо псевдокода мы работаем с реальным кодом, чтобы программы можно было быстро применить на практике. Наши программы написаны на Java, но в таком стиле, что бо́льшую часть кода можно использовать для разработки реализаций на других современных языках программирования.

- **Типы данных.** Мы применяем современный стиль программирования, основанный на абстракции данных, когда алгоритмы и соответствующие им структуры данных инкапсулированы вместе.
- **Приложения.** Каждая глава содержит подробное описание приложений, в которых описываемые алгоритмы играют важную роль. Это приложения наподобие задач из физики и молекулярной биологии, разработки компьютеров и систем, а также такие знакомые каждому задачи, как сжатие данных и поиск во всемирной сети.
- **Научный подход.** Мы стараемся разрабатывать математические модели для описания производительности алгоритмов, используем эти модели для выдвижения гипотез о производительности, а затем проверяем гипотезы, выполняя алгоритмы в контекстах, приближенных к действительности.
- **Охват.** Мы рассматриваем базовые абстрактные типы данных, алгоритмы сортировки, алгоритмы поиска, обработку графов и обработку строк. Весь материал подается в алгоритмическом контексте, с описанием структур данных, парадигм построения алгоритмов, сведения и моделей решения задач. Мы рассказываем как о классических методах, которые изучаются с 1960-х годов, так и о новых методах, разработанных в самые последние годы.

Наша основная цель — познакомить с наиболее важными на сегодняшний день алгоритмами максимально широкую аудиторию. Эти алгоритмы, как правило, являются хитроумными творениями, которые выражаются в десятке-двух строках кода. Все вместе они представляют мощь невероятного охвата. Они позволили создавать вычислительные артефакты, решать научные задачи и разрабатывать коммерческие приложения, которые не могли бы работать без них.

Сайт книги

Важным компонентом книги является ее связь с сайтом `algs4.cs.princeton.edu`. Этот сайт доступен всем и содержит значительный объем материала об алгоритмах и структурах данных для преподавателей, студентов и практиков.

Краткое онлайновое описание

Сайт книги содержит краткое содержание книги в той же самой структуре, но со ссылками, облегчающими навигацию по материалу.

Полные реализации

Весь код, приведенный в данной книге, доступен на ее сайте в форме, пригодной для разработки программ. Приведены и многие другие реализации, в том числе расширенные реализации и усовершенствования, описанные в книге, ответы на некоторые упражнения и клиентский код для различных приложений. Основная цель кода — тестирование алгоритмов в контексте осмысленных приложений.

Упражнения и ответы

Сайт книги вдобавок к упражнениям, приведенным в книге, содержит тренировочные упражнения (ответы к которым доступны по щелчку), огромное количество примеров, иллюстрирующих богатство материала, упражнения по программированию с решениями в виде готового кода и исследовательские задачи.

Динамические визуализации

В печатную книгу невозможно вставить динамические модели, но на веб-сайте имеется много реализаций, которые используют графический класс для представления великолепных визуальных демонстраций применений алгоритмов.

Материалы для занятий

С материалом, приведенным в книге и на сайте, непосредственно связан полный набор лекционных слайдов. Там же имеется полный набор заданий по программированию с инструкциями, тестовыми данными и материалами для занятий.

Ссылки на сопутствующий материал

Сотни ссылок позволяют ознакомиться с сопутствующей информацией о приложениях и с источниками для дальнейшего изучения алгоритмов.

Нашей целью при создании этого материала было обеспечение дополнительных подходов к идеям, изложенным в книге. Мы считаем, что книгу следует читать при первоначальном изучении конкретных алгоритмов или для получения обшей картины, а сайт лучше использовать как справочник при программировании или как исходную точку при поиске более подробной информации в Интернете.

Использование в учебном плане

Эта книга задумана как учебник для второго курса обучения вычислительной технике. Она содержит полное изложение базового материал и является великолепным пособием для студентов, которые хотят набраться опыта в программировании, количественных оценках и решении задач. Обычно в качестве необходимой подготовки достаточно одного курса вычислительной техники: книга предназначена для всех, кто знаком с современными языками программирования и с основными компонентами современных компьютерных систем.

Алгоритмы и структуры данных записаны на Java, но в таком стиле, который понятен и людям, хорошо знакомым с другими современными языками. Мы используем современные абстракции языка Java (включая обобщенные типы), но не связываемся с экзотическими особенностями этого языка.

Большая часть математического материала, сопровождающего аналитические результаты, либо понятна без объяснений, либо объявлена как выходящая за рамки книги. Поэтому для чтения книги обычно не требуется специальная математическая подготовка, хотя опыт математических вычислений, несомненно, будет полезен. Примеры применения взяты из вводного материала по различным наукам и поэтому также не требуют пояснений.

Изложенный материал представляет собой фундамент для любого студента, который хочет освоиться в вычислительной технике, электротехнике или исследовании операций, и пригодится любому, кто интересуется наукой, математикой или техникой.

Контекст

Книга задумана как продолжение вводного учебника *An Introduction to Programming in Java: An Interdisciplinary Approach*, который представляет собой основательное введение в программирование на языке Java. Вместе эти две книги могут составлять двух- или трехсеместровое введение в вычислительную технику, которое даст любому студенту базу для успешного освоения вычислительных методов в любой области науки, техники или социального направления.

Основой для большей части материала этой книги составляют книги Седжвика “Фундаментальные алгоритмы”. По своему духу данная книга ближе к первому и второму изданиям “Алгоритмов”, но текст обогащен десятилетиями опыта в преподавании и изучении материала. Относительно недавно вышедшая книга Седжвика “Алгоритмы на C/C++/Java, 3-е издание” скорее пригодна как справочник или учебник для более продвинутого курса. Данная же книга специально задумана как основа односеместрового курса для студентов первого или второго курса колледжа, как современное введение в основы и как справочник для работающих программистов.

Благодарности

Эта книга развивается уже почти 40 лет, поэтому немыслимо полностью перечислить всех причастных к ее выпуску. Первые издания книги содержат десятки имен, в том числе (в алфавитном порядке) Эндрю Аппеля (Andrew Appel), Трину Эйвери (Trina Avery), Марка Брауна (Marc Brown), Лин Дюпре (Lyn Dupré), Филиппа Флажоле (Philippe Flajolet), Тома Фримена (Tom Freeman), Дэйва Хэнсона (Dave Hanson), Джанет Инсерпай (Janet Incerpi), Майка Шидловски (Mike Schidlowsky), Стива Саммита (Steve Summit) и Криса ван Вика (Chris Van Wyk). Все они заслуживают благодарности, даже если они приложили руку десятки лет назад. Выпуская это четвертое издание, мы говорим спасибо сотням студентов Принстонского университета и нескольких других учреждений, которые вытерпели несколько различных предварительных версий данной работы, а также читателям со всего мира — за их комментарии и поправки через сайт книги.

Мы благодарим за поддержку Принстонский университет, за его неуклонное стремление к совершенству в обучении и изучении, что является фундаментом для формирования данной книги.

Питер Гордон (Peter Gordon) дал множество мудрых советов почти с самого начала разработки этой книги, кроме того, он мягко настаивал на принципе “возврата к основам”. В отношении данного четвертого издания мы благодарим Барбару Вуд (Barbara Wood) за ее тщательное и профессиональное редактирование, Джулию Нагил (Julie Nahil) за руководство выпуском и многих других людей из издательства Pearson за их вклад в разработку и маркетинг книги. Все отлично уложились в довольно плотный график, при этом ни на йоту не пожертвовав качеством результата.

Роберт Седжвик
Кевин Уэйн
Принстон, шт. Нью-Джерси
январь 2011 г.

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо, либо просто посетить наш веб-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг.

Наши координаты:

E-mail: `info@williamspublishing.com`
WWW: `http://www.williamspublishing.com`

Информация для писем из:

России: 127055, г. Москва, ул. Лесная, д. 43, стр. 1
Украины: 03150, Киев, а/я 152

Глава 1

Основные понятия

Цель этой книги — изучение множества важных и полезных *алгоритмов*, т.е. методов решения задач, которые пригодны для реализации на компьютере. Алгоритмы неотделимы от *структур данных* — схем для организации данных, которые позволяют выполнять эффективную обработку этих данных алгоритмами. В данной главе вы познакомитесь с базовыми средствами, которые необходимы для изучения алгоритмов и структур данных.

Вначале мы опишем нашу *базовую модель программирования*. Все наши программы реализованы с помощью небольшого подмножества языка программирования Java и нескольких наших собственных библиотек для ввода-вывода и статистических расчетов. Раздел 1.1 содержит сводку по конструкциям и компонентам языка и библиотекам, которые используются в данной книге.

Потом мы рассмотрим *абстракцию данных* и определим *абстрактные типы данных* (АТД), которые делают возможным модульное программирование. В разделе 1.2 мы познакомимся с процессом реализации АТД на Java с помощью *интерфейса прикладного программирования* (applications programming interface — API), а затем воспользуемся механизмом классов Java для разработки реализации, которая может использоваться в клиентском коде.

После этого в качестве важных и полезных примеров мы рассмотрим три фундаментальных АТД: *контейнер*, *очередь* и *стек*. В разделе 1.3 будут описаны API-интерфейсы и реализации контейнеров, очередей и стеков с помощью массивов, массивов с переменным размером и связных списков — они послужат в качестве моделей и отправных точек для реализации алгоритмов во всей книге.

Производительность является основной темой при изучении алгоритмов. В разделе 1.4 описан наш подход к анализу производительности алгоритмов. В основе этого анализа лежит *научный метод*: мы выдвигаем гипотезы о производительности, создаем математические модели, а затем выполняем эксперименты с целью их проверки; при необходимости этот процесс повторяется.

В конце главы нас ждет учебный пример; в нем будут рассмотрены решения задачи *связности*, в которой применяются алгоритмы и структуры данных, реализующие классический АТД *объединения-поиска*.

Алгоритмы

При написании компьютерных программ мы обычно реализуем *метод*, который уже был разработан ранее для решения какой-то задачи. Такой метод часто не зависит от конкретного используемого языка программирования и одинаково пригоден для многих компьютеров и многих языков программирования. Именно метод, а не компьютерная программа, описывает шаги, которые нужно выполнить для решения задачи. Термин *алгоритм* применяется в вычислительной технике для описания конечного, детерминированного и эффективного метода решения задачи, который годится для реализации в виде компьютерной программы. Алгоритмы представляют собой основные объекты изучения в данной области.

Алгоритм можно определить, описав процедуру для решения задачи на естественном языке или в виде компьютерной программы, которая реализует эту процедуру.

На рис. 1.0.1 приведены оба этих варианта для *алгоритма Евклида*, который был разработан более 2300 лет назад для нахождения наибольшего общего делителя двух чисел. Если вы еще не знакомы с алгоритмом Евклида, рекомендуем проработать уп-

ражнения 1.1.24 и 1.1.25, возможно, после прочтения раздела 1.1. В данной книге для описания алгоритмов используются компьютерные программы. Одна из важных причин для этого — так легче проверить, что они действительно конечны, детерминированы и эффективны. Но все же необходимо понимать, что программа на конкретном языке представляет собой лишь один из способов выражения алгоритма. Многие алгоритмы, приведенные в данной книге, за последние десятилетия уже написаны на множестве языков программирования — это подтверждает мысль, что каждый алгоритм является методом, который можно реализовать на любом компьютере и с помощью любого языка программирования.

Описание на русском языке

Чтобы вычислить наибольший общий делитель двух неотрицательных целых чисел p и q, нужно сделать следующее. Если q равно 0, берем в качестве ответа p. Если не равно 0, разделим p на q с вычислением остатка r. Ответом является наибольший общий делитель q и r.

Описание на языке Java

```
public static int gcd(int p, int q)
{
   if (q == 0) return p;
   int r = p % q;
   return gcd(q, r);
}
```

Рис. 1.0.1. *Алгоритм Евклида*

Большинство полезных алгоритмов требуют организации данных, участвующих в вычислениях. Такая организация называется *структурами данных*, и эти структуры также являются основными объектами изучения в вычислительной технике. Алгоритмы и структуры данных работают в тесном взаимодействии. В настоящей книге мы придерживаемся мнения, что структуры данных существуют как побочные или конечные продукты алгоритмов, и их изучение необходимо для понимания алгоритмов. Простые алгоритмы могут потребовать сложных структур данных, и наоборот, сложные алгоритмы могут использовать простые структуры данных. Мы исследуем в книге свойства многих структур данных, так что саму книгу можно было бы назвать “Алгоритмы и структуры данных”.

Когда мы используем компьютер для решения задач, то обычно сталкиваемся с несколькими возможными подходами. Для небольших задач почти нет разницы, какой из подходов использовать, если они все правильно решают задачу. Однако в случае больших задач (или в приложениях, где требуется решать много маленьких задач) быстро осознается необходимость методов, которые эффективно используют время и память.

Основной причиной изучения алгоритмов является то, что эта дисциплина в принципе позволяет значительно экономить ресурсы — вплоть до того, что становится возможным выполнять задачи, которые иначе были бы недоступны. В приложении, которое обрабатывает миллионы объектов, совсем не редкость ускорение работы в миллионы раз с помощью отточенного алгоритма. Мы неоднократно встретимся с такими примерами на протяжении книги. А вложение дополнительных денег или усилий на приобретение и установку нового компьютера может ускорить работу программы раз в 10 или, скажем, в 100. Тщательная разработка алгоритма — невероятно эффективная часть процесса решения большой задачи в любой прикладной области.

При разработке большой или сложной компьютерной программы следует потратить много усилий на понимание и четкое определение решаемой задачи, управление ее сложностью и разложение на меньшие подзадачи, которые несложно реализовать. Зачастую после такого разложения нужные алгоритмы реализуются элементарно. Однако в большинстве случаев имеется несколько алгоритмов, и выбор одного из них крайне важен, т.к. значительная часть системных ресурсов будет потрачена на выполнение этих алгоритмов. Именно такие алгоритмы и будут в основном рассматриваться в настоящей книге. Мы изучаем фундаментальные алгоритмы, которые позволяют решать сложные задачи в широком диапазоне прикладных областей.

Использование чужих разработок в компьютерных системах постоянно ширится, поэтому следует ожидать не только того, что нам придется *использовать* значительную часть алгоритмов из этой книги, но и того, что *реализовывать* придется лишь небольшую их часть. Например, библиотеки Java содержат реализации огромного количества фундаментальных алгоритмов. Однако реализация простых вариантов базовых алгоритмов поможет нам лучше понять их и поэтому более эффективно использовать, а также осмысленно настраивать сложные версии из библиотек. И, что более важно, зачастую возникает необходимость в самостоятельной реализации базовых алгоритмов. Обычно такая необходимость возникает, если приходится начинать работу в новых вычислительных средах (как аппаратных, так и программных) с новыми возможностями, которые не учитываются в старых реализациях. В данной книге в основном описываются наиболее простые и разумные реализации наилучших алгоритмов. Мы будем уделять самое пристальное внимание кодированию критических частей алгоритмов и каждый раз указывать, где будет уместны усилия по низкоуровневой оптимизации.

Выбор оптимального алгоритма для конкретной задачи может оказаться отнюдь не простой проблемой, возможно, с привлечением сложного математического анализа. Отрасль вычислительной техники, которая изучает такие вопросы, называется *анализом алгоритмов*. Для многих алгоритмов, которые мы будем рассматривать, аналитически доказана отличная теоретическая производительность, но для других имеются лишь экспериментальные данные об их поведении. Наша основная цель — изучение приемлемых алгоритмов для важных задач, но мы придаем важное значение и сравнению производительности методов. Мы не будем рекомендовать какой-либо алгоритм к применению, если совершенно непонятно, какие ресурсы ему потребуются, и поэтому будем стараться всякими способами получить данные об ожидаемой производительности алгоритмов.

Краткий обзор тем

В качестве обзора мы опишем основные части книги, основные рассматриваемые темы и направления, в которых мы будем изучать материал. Этот набор тем подобран таким образом, чтобы затронуть как можно больше фундаментальных алгоритмов. Некоторые из рассматриваемых областей представляют собой ключевые области вычислительной техники, которые мы будем подробно изучать, чтобы освоить широко применяемые базовые алгоритмы. Другие алгоритмы взяты из более сложных тем вычислительной техники и связанных с ними областей. Рассматриваемые нами алгоритмы являются результатом десятилетий исследований и разработок, и они продолжают играть важную роль в постоянно растущем применении компьютерных вычислений.

Основные понятия (глава 1) в контексте данной книги — это методология и базовые принципы, которые мы будем использовать для реализации, анализа и сравнения алгоритмов. Мы рассмотрим нашу модель программирования на Java, абстракцию данных,

базовые структуры данных, абстрактные типы данных для коллекций данных, методы анализа производительности алгоритмов и учебный пример.

Сортировка (глава 2). Алгоритмы для упорядочивания массивов имеют фундаментальную важность. Мы довольно глубоко рассмотрим множество алгоритмов: сортировка вставками, сортировка выбором, сортировка Шелла, быстрая сортировка, сортировка слиянием и пирамидальная сортировка. Кроме того, мы изучим алгоритмы для нескольких схожих задач: очереди с приоритетами, выбор и слияние. Многие из этих алгоритмов будут использованы в качестве фундамента для построения других алгоритмов в последующих главах книги.

Поиск (глава 3). Не меньшую фундаментальную важность имеют и алгоритмы для нахождения конкретных элементов в больших коллекциях элементов. Мы рассмотрим простые и более сложные методы поиска: двоичные деревья поиска, сбалансированные деревья поиска и хеширование, изучим взаимосвязи этих методов и сравним их производительность.

Графы (глава 4) — это наборы объектов и связей между ними, возможно, с весами и ориентацией этих связей. Графы представляют собой удобные модели для огромного количества сложных и важных задач, и разработка алгоритмов для обработки графов является одной из основных областей изучения. Мы рассмотрим поиск в глубину, поиск в ширину, задачи связности и несколько других алгоритмов и приложений: алгоритмы Крускала и Прима для поиска минимального остовного дерева, а также алгоритмы Дейкстры и Беллмана-Форда для определения кратчайших путей.

Строки (глава 5) — важный тип данных в современных вычислительных приложениях. Мы рассмотрим ряд методов для обработки последовательностей символов. Сначала это будут более быстрые алгоритмы для сортировки и поиска в случае, когда ключи представляют собой строки. Затем мы рассмотрим поиск подстрок, сравнение с шаблонами регулярных выражений и алгоритмы сжатия данных. Здесь также будет изложено краткое введение в более сложные темы на примере разбора некоторых элементарных задач, которые важны сами по себе.

Контекст (глава 6) поможет связать материал, изложенный в книге, с несколькими другими продвинутыми областями изучения: научные вычисления, исследование операций и теория вычислений. Мы рассмотрим моделирование на основе событий, B-деревья, суффиксные массивы, максимальные потоки и другие сложные темы — только начала, но этого будет достаточно для ознакомления с интересными более сложными областями изучения, где алгоритмы играют критически важную роль. И в конце будут описаны задачи поиска, приведения и NP-полноты, которые позволят понять теоретические основы изучения алгоритмов и взаимосвязь с материалом, изложенным в данной книге.

Изучение алгоритмов — интересное и захватывающее занятие, потому что это новая область знаний (почти все изучаемые нами алгоритмы имеют возраст менее 50 лет, а некоторые открыты вообще недавно), но уже с богатыми традициями (несколько алгоритмов известны сотни лет). Постоянно совершаются новые открытия, но полностью поняты лишь немного алгоритмов. В этой книге мы рассмотрим как хитроумные, сложные и трудные алгоритмы, так и элегантные, простые и легкие. Наша задача — понять алгоритмы из первой категории и оценить достоинства второй категории в контексте научных и коммерческих приложений. При этом мы познакомимся с множеством полезных средств и разовьем стиль *алгоритмического мышления*, которое очень пригодится при решении вычислительных задач в будущем.

1.1. Базовая модель программирования

Наше изучение алгоритмов основано на их реализации в виде *программ*, записанных на языке программирования Java. На это имеется несколько причин.

- Наши программы представляют собой лаконичные, элегантные и полные описания алгоритмов.
- Программы можно запускать для изучения свойств алгоритмов.
- Алгоритмы можно сразу применять в приложениях.

Это важные и серьезные преимущества по сравнению с описаниями алгоритмов на естественном языке.

Потенциальный недостаток этого подхода — приходится работать с конкретным языком программирования, и, возможно, будет трудно отделить саму идею алгоритма от деталей ее реализации. Наши реализации написаны так, чтобы свести к минимуму эту трудность: мы будем использовать программные конструкции, которые присутствуют в большинстве современных языков и необходимы для адекватного описания алгоритмов.

Мы будем использовать лишь небольшое подмножество Java. После того как мы формально определим необходимое подмножество, вы увидите, что в него входит относительно немного конструкций Java, и мы выделяем те из них, которые имеются во многих современных языках программирования. Представленный в книге код полностью завершен, и мы ожидаем, что вы загрузите его и выполните — на наших или своих тестовых данных.

Программные конструкции, библиотеки программного обеспечения и возможности операционной системы, которые мы будем применять для реализации и описания алгоритмов, мы называем нашей *моделью программирования*. Эта модель будет полностью описана в этом разделе и разделе 1.2. Описание не требует дополнительных источников и в основном предназначено для документирования и в качестве справочного материала для понимания любого кода из этой книги. Описываемая здесь модель — это та же модель, что и представленная в книге *An Introduction to Programming in Java: An Interdisciplinary Approach* (Addison Wesley, 2007 г.), но там она изложена более развернуто.

На рис. 1.1.1 приведен пример полной Java-программы, которая иллюстрирует многие основные черты нашей модели программирования. Мы используем этот код просто для обсуждения свойств языка, а его смысл будет описан позже (в нем реализован классический алгоритм, известный как *бинарный поиск*, и он проверяется в приложении, которое называется *фильтрацией по белому списку*). Мы предполагаем, что вы имеете опыт программирования на каком-то современном языке, так что вы, видимо, сразу узнаете многие его черты в данном коде. В пояснениях приведены ссылки на страницы, которые помогут вам быстро найти ответы на возможные вопросы. Наш код оформлен в определенном стиле, и мы будем стараться придерживаться его в использовании различных идиом и конструкций языка Java, поэтому даже опытным программистам на Java рекомендуется внимательно прочитать данный раздел.

```
import java.util.Arrays;

public class BinarySearch
{
   public static int rank(int key, int[] a)
   {
      int lo = 0;
      int hi = a.length - 1;
      while (lo <= hi)
      {
         int mid = lo + (hi - lo) / 2;
         if      (key < a[mid]) hi = mid - 1;
         else if (key > a[mid]) lo = mid + 1;
         else                   return mid;
      }
      return -1;
   }

   public static void main(String[] args)
   {

      int[] whitelist = In.readInts(args[0]);

      Arrays.sort(whitelist);

      while (!StdIn.isEmpty())
      {
         int key = StdIn.readInt();
         if (rank(key, whitelist) == -1)
            StdOut.println(key);
      }
   }
}
```



```
% java BinarySearch largeW.txt < largeT.txt
499569
984875
...
```

Рис. 1.1.1. Структура Java-программы и ее вызов из командной строки

Базовая структура Java-программы

Java-программа (*класс*) — это либо *библиотека статических методов* (функций), либо *определение типа данных*. Для создания библиотек статических методов и определений типов данных мы используем следующие семь компонентов, которые составляют фундамент программирования на Java и многих других современных языках.

- *Примитивные типы данных* в точности определяют в компьютерной программе значение таких терминов, как *целое число*, *вещественное число* и *логическое значение*. В определение входят набор возможных значений и *операции* с этими значениями, которые можно объединять для получения *выражений* — например, математических выражений, которые определяют значения.
- *Операторы* позволяют определять вычисления с помощью создания значений и присваивания их *переменным*, управлять потоком выполнения или вызывать побочные эффекты. Мы будем использовать шесть типов операторов: *объявления*, *присваивания*, *условные операторы*, *циклы*, *вызовы* и *возвраты*.
- *Массивы* позволяют работать с несколькими значениями одного типа.
- *Статические методы* позволяют обособить и повторно использовать код и разрабатывать программы в виде набора независимых модулей.
- *Строки* представляют собой последовательности символов. Некоторые операции над ними встроены в Java.
- *Ввод-вывод* позволяет программам взаимодействовать с внешним миром.
- *Абстракция данных* расширяет обособление и повторное использование и позволяет определять непримитивные типы данных, поддерживая объектно-ориентированное программирование.

В текущем разделе мы рассмотрим по порядку лишь первые шесть из этих компонентов. Абстракция данных — тема следующего раздела.

Для выполнения Java-программы необходимо взаимодействие с операционной системой или средой разработки программ. Для ясности и краткости мы будем описывать такие действия в терминах *виртуального терминала*, где мы взаимодействуем с программами, вводя команды в систему. На сайте книги содержится информация об использовании виртуального терминала в вашей системе и о многих продвинутых средах разработки ПО, доступных в современных системах.

Например, класс `BinarySearch` на рис. 1.1.1 содержит два статических метода: `rank()` и `main()`. Первый статический метод, `rank()`, состоит из четырех операторов: двух объявлений, цикла (который сам содержит присваивание и два условных оператора) и возврата. Второй метод, `main()`, состоит из трех операторов: объявления, вызова метода и цикла (который содержит присваивание и условный оператор).

Чтобы вызвать Java-программу, ее вначале нужно *скомпилировать* с помощью команды `javac`, а затем *запустить* (или *выполнить*) с помощью команды `java`.

Например, для выполнения класса `BinarySearch` потребуется ввести команду `javac BinarySearch.java` (которая создает файл `BinarySearch.class` с низкоуровневой версией Java-программы — *байт-код* Java). После этого необходимо ввести команду `java BinarySearch` (с последующим именем файла) для передачи управления этой байтовой версии программы. Теперь, чтобы образовать прочный фундамент, который позволит разобраться в эффекте этих действий, нам нужно подробно рассмотреть примитивные

типы данных и выражения, различные операторы языка Java, массивы, статические методы строки и ввод-вывод.

Примитивные типы данных и выражения

Тип данных — это множество значений и множество операций над этими значениями. Вначале мы рассмотрим следующие четыре *примитивных* типа данных, которые лежат в основе языка Java:

- *целые числа* с арифметическими операциями (`int`);
- *вещественные числа*, также с арифметическими операциями (`double`);
- *логические значения* — множество значений {*истина*, *ложь*} с логическими операциями (`boolean`);
- *символы* — алфавитно-цифровые и другие символы (`char`).

А теперь рассмотрим механизмы для задания значений и указания операций для всех этих типов.

Java-программа работает с *переменными*, которые можно именовать и различать по их именам — *идентификаторам*. Каждая переменная приписана к какому-то типу данных и хранит одно из допустимых значений этого типа данных. В Java-коде используются *выражения*, похожие на обычные математические выражения, в которых применяются операции, связанные с каждым типом. Для примитивных типов используются идентификаторы для обращения к переменным, символы *операций* наподобие `+ - * /` для указания операций, *литералы* вроде `1` или `3.14` для задания значений и выражения наподобие `(x + 2.236)/2` для указания операций над значениями. Выражения предназначены для определения одного из значений какого-то типа данных.

Таблица 1.1.1. Базовые строительные блоки для Java-программ

Термин		Примеры	Определение
Примитивный тип данных		`int` `double` `boolean` `char`	Множество значений и множество операций над этими значениями (встроены в язык Java)
Идентификатор		`a abc Ab$ a_b` `ab123 lo hi`	Последовательность букв, цифр, _ и $; в начале не должна быть цифра
Переменная		[*любой идентификатор*]	Именует значение типа данных
Операция		`+ - * /`	Именует операцию типа данных
Литерал	`int` `double` `boolean` `char`	`1 0 -42` `2.0 1.0e-15 3.14` `true false` `'a' '+' '9' '\n'`	Представление значения в исходном коде
Выражение	`int` `double` `boolean`	`lo + (hi - lo)/2` `1.0e-15 * t` `lo <= hi`	Литерал, переменная или последовательность операций над литералами и/или переменными, которая дает значение

Чтобы определить тип данных, достаточно задать значения и множество операций над этими значениями. Для типов данных `int`, `double`, `boolean` и `char` из Java эта информация приведена в табл. 1.1.2. Описанные типы данных похожи на базовые типы данных, которые имеются во многих языках программирования. Для типов `int` и `double` операции представляют собой знакомые всем арифметические операции, для типа `boolean` это также широко известные логические операции. Важно понимать, что операции +, −, * и / *перегружены*: одним и тем же символом обозначаются операции для нескольких различных типов и зависит от контекста. Основное свойство этих примитивных операций формулируется так: *операция со значениями одного типа дает в результате значение этого же типа*. Это правило подчеркивает мысль, что мы часто работаем с приближительными значениями, т.к. значение, определенное выражением, не всегда может быть значением данного типа. Например, выражение `5/3` имеет значение `1`, а `5.0/3.0` — значение, очень близкое к `1.66666666666667`, но ни одно из них не равно в точности 5/3.

Таблица 1.1.2 далеко не полна; некоторые дополнительные операции и различные исключительные ситуации, которые иногда приходится учитывать, обсуждаются в разделе "Вопросы и ответы" в конце текущего раздела.

Таблица 1.1.2. Примитивные типы данных в Java

Тип	Множество значений	Операции	Типичные выражения	
			выражение	значение
`int`	Целые числа от -2^{31} до $2^{31}-1$ (32-разрядное двоичное дополнение)	+ (сложение)	`5 + 3`	`8`
		− (вычитание)	`5 - 3`	`2`
		* (умножение)	`5 * 3`	`15`
		/ (деление)	`5 / 3`	`1`
		% (остаток)	`5 % 3`	`2`
`double`	Вещественные числа с двойной точностью (64-разрядный стандарт IEEE 754)	+ (сложение)	`3.141 - .03`	`3.111`
		− (вычитание)	`2.0 - 2.0e-7`	`1.9999998`
		* (умножение)	`100 * .015`	`1.5`
		/ (деление)	`6.02e23 / 2.0`	`3.01e23`
`boolean`	`true` или `false`	&& (и)	`true && false`	`false`
		|| (или)	`false || true`	`true`
		! (не)	`!false`	`true`
		^ (исключающее или)	`true ^ true`	`false`
`char`	Символы (16-разрядные)	[*арифметические операции, используются редко*]		

Выражения

Как видно в табл. 1.1.2, обычно выражения имеют *инфиксную* форму: литерал (или выражение), за ним знак операции, а за ним другой литерал (или другое выражение). Если выражение содержит более одной операции, порядок их выполнения зачастую играет важную роль. Поэтому частью спецификации языка Java являются следующие соглашения по *приоритету* (старшинству): операции * и / (и %) имеют приоритет больший, чем (записанные ранее) операции + и −; а среди логических операций наивысший приоритет имеет !, затем && и, наконец, ||. Как правило, операции одного приоритета выполняются слева направо. Как и в обычных арифметических выражениях, порядок выполнения операций можно изменить с помощью скобок.

Правила старшинства операций могут слегка меняться в различных языках, поэтому мы применяем скобки и вообще стараемся избежать в нашем коде зависимости от правил старшинства.

Преобразование типов

Тип чисел автоматически повышается до более объемлющего типа, если не происходит потери информации. Например, в выражении `1 + 2.5` значение `1` преобразуется в `1.0`, а затем вычисление выражения дает значение `3.5`. *Приведение* — это имя типа в скобках в составе выражения, оно означает директиву преобразовать следующее за ним выражение в указанный тип. Например, `(int) 3.7` равно `3`, а `(double) 3` равно `3.0`. Обратите внимание, что приведение к типу `int` означает усечение, а не округление: правила приведения в сложных выражениях могут быть запутанными, поэтому приведения следует использовать редко и осторожно. Лучше всего использовать выражения, которые содержат литералы и/или переменные одного типа.

Сравнения

Следующие операции сравнивают два значения одного типа и выдают логическое значение: *равно* (==), *не равно* (!=), *меньше* (<), *меньше или равно* (<=), *больше* (>) и *больше или равно* (>=). Эти операции называются операциями *со смешанными типами*, т.к. их значение имеет тип `boolean`, а типы сравниваемых значений могут быть другими. Выражение с логическим значением называется *логическим выражением*. Как мы вскоре увидим, такие выражения являются важными компонентами условных операторов и операторов цикла.

Другие примитивные типы

Тип `int` в Java по построению может иметь 2^{32} различных значений, поэтому он может быть представлен в 32-разрядном машинном слове (сейчас во многих компьютерах имеются 64-разрядные слова, но 32-разрядный `int` никуда не делся). Аналогично, стандарт типа `double` задает 64-разрядное представление. Эти размеры типов данных вполне пригодны для типичных приложений, где используются целые и вещественные числа. Для большей гибкости в Java предусмотрены пять дополнительных примитивных типов данных:

- 64-разрядные целые числа с арифметическими операциями (`long`);
- 16-разрядные целые числа с арифметическими операциями (`short`);
- 16-разрядные символы с арифметическими операциями (`char`);
- 8-разрядные целые числа с арифметическими операциями (`byte`);
- 32-разрядные вещественные числа с одинарной точностью, также с арифметическими операциями (`float`).

В нашей книге мы чаще всего будем использовать арифметические типы `int` и `double`, поэтому остальные типы (очень похожие) здесь не рассматриваются.

Операторы

Java-программа состоит из *операторов*, которые для определения процесса вычисления создают переменные и работают с ними, присваивают переменным значения нужных типов данных и управляют потоком выполнения таких операций. Операторы часто оформлены в блоки — последовательности операторов в фигурных скобках.

- *Объявления* создают переменные указанного типа и присваивают им имена — идентификаторы.
- *Операторы присваивания* связывают значения типов данных (определенные выражениями) с переменными. В языке Java имеется также несколько идиом *неявного присваивания* для изменения значения некоторой переменной относительно текущего значения этой переменной — например, увеличение значения целой переменной на единицу.
- *Условные операторы* предназначены для простого изменения потока выполнения, когда в зависимости от заданного условия выполняется один из двух блоков.
- *Циклы* предназначены для более существенного изменения потока выполнения: операторы в блоке выполняются, пока верно заданное условие.
- *Вызовы* и *возвраты* относятся к статическим методам, которые предоставляют еще один способ изменения потока выполнения и организации кода.

Программа представляет собой последовательность операторов объявления, присваивания, условных операторов, операторов цикла, вызова и возврата. Программы обычно имеют *вложенную* структуру: оператор среди других операторов блока в условном операторе или цикле сам может быть условным оператором или циклом. Например, на рис. 1.1.1 цикл `while` в методе `rank()` содержит условный оператор `if`. Теперь мы рассмотрим все эти виды операторов.

Объявления

Оператор *объявления* связывает имя переменной с типом во время компиляции (табл. 1.1.3). В Java необходимо использовать объявления для указания имен и типов переменных, чтобы явно определиться со всеми вычислениями. Java называется *строго типизированным* языком, т.к. компилятор Java проверяет согласованность типов (например, он не разрешает умножить значение `boolean` на `double`). Объявления могут находиться в любом месте, но до первого использования переменной — чаще всего как раз *в* месте первого использования. *Область видимости* (или *область действия*) переменной — это часть программы, где она определена. Обычно область видимости переменной состоит из операторов, которые следуют за объявлением этой переменной в том же блоке, что и объявление.

Присваивания

Оператор *присваивания* связывает некоторое значение типа данных (определенное выражением) с некоторой переменной. Запись `c = a + b` в языке Java означает не математическое равенство, а действие: в переменную `c` заносится значение, равное сумме значений `a` и `b`. Правда, сразу после выполнения оператора присваивания `c` математически равно `a + b`, но главное в этом операторе — изменение значения `c` (при необходимости). В левой части оператора присваивания должна находиться одна переменная, а в правой части может быть произвольное выражение, выдающее значение нужного типа.

Условные операторы

В большинстве вычислений для различных входных данных требуется выполнять различные действия. Одним из способов выражения этих различий в Java является оператор `if`:

```
if (<логическое выражение>) { <операторы блока> }
```

В этом выражении введена формальная запись, которая называется *шаблоном*; мы будем иногда применять ее для описания формата конструкций языка Java. В угловые скобки (< >) помещается конструкция, уже определенная ранее — это означает, что в этом месте можно использовать любой экземпляр такой конструкции. В данном случае *<логическое выражение>* означает выражение, которое имеет логическое значение (например, в результате операции сравнения), а *<операторы блока>* означает последовательность операторов Java. Хорошо бы иметь формальные определения для конструкций *<логическое выражение>* и *<операторы блока>*, но мы не будем вдаваться в такие подробности. Смысл оператора `if` понятен без объяснений: операторы в блоке должны выполняться тогда и только тогда, когда логическое выражение равно `true`.

Оператор `if-else` выглядит так:

```
if (<логическое выражение>) { <операторы блока> }
else                         { <операторы блока> }
```

Он позволяет выбрать один из двух альтернативных блоков операторов.

Циклы

Многие вычисления по своей сути требуют многократных повторений каких-то действий. Простейшая конструкция языка Java для выполнения таких вычислений имеет следующий формат:

```
while (<логическое выражение>) { <операторы блока> }
```

Оператор `while` выглядит точно так же, как и оператор `if` (разница только в ключевом слове `while` вместо `if`), но его смысл совсем другой. Это указание компьютеру выполнять следующее: если логическое выражение равно `false`, не делать ничего; если логическое выражение равно `true`, выполнить последовательность операторов в блоке (как и в операторе `if`), но затем снова проверить значение логического выражения, снова выполнить последовательность операторов в блоке, если это значение равно `true`, и продолжать так, пока логическое выражение равно `true`. Операторы в блоке в этом случае называются *телом* цикла.

Прерывание и продолжение

В некоторых ситуациях требуется несколько более сложное управление потоком вычислений, чем позволяют простые операторы `if` и `while`. Поэтому в Java имеются два дополнительных оператора для использования в циклах `while`:

- оператор `break`, который приводит к немедленному выходу из цикла;
- оператор `continue`, который приводит к немедленному началу следующей итерации цикла.

Мы редко будем использовать эти операторы в настоящей книге (а многие программисты и вовсе их не используют), но иногда они существенно упрощают код.

Сокращенные обозначения

Многие вычисления можно записать несколькими способами, а мы хотим получить ясный, элегантный и эффективный код. В таком коде часто применяются перечисленные ниже широко распространенные сокращения (которые присутствуют во многих языках, не только в Java).

Инициализирующие объявления

Можно совместить объявление с присваиванием и выполнить инициализацию непосредственно в момент объявления (создания). Например, код `int i = 1;` создает переменную типа `int` с именем `i` *и* присваивает ей начальное значение `1`. Рекомендуется применять такую запись вблизи первого использования переменной (чтобы ограничить область ее действия).

Неявные присваивания

Если необходимо изменить значение переменной относительно ее текущего значения, то доступны следующие сокращения.

- Операции инкремента/декремента (увеличения/уменьшения на единицу). Запись `++i` эквивалентна `i = i + 1` и обе они имеют значение `i+1` в выражении. Аналогично `--i` эквивалентно `i = i - 1`. Код `i++` и `i--` выполняется так же, но значение выражения берется ***перед*** выполнением увеличения или уменьшения, а не после.
- Другие составные операции. Добавление бинарной операции перед знаком присваивания = эквивалентно использованию переменной слева в качестве первого операнда. Например, код `i/=2;` эквивалентен коду `i = i/2;` Оператор `i += 1;` дает тот же эффект, что и `i = i + 1;` (и `i++`).

Блоки из одного оператора

Если блок операторов в условном операторе или цикле содержит только один оператор, фигурные скобки вокруг него не обязательны.

Запись `for`

Многие циклы выполняются по следующей схеме: вначале индексная переменная инициализируется некоторым значением, а затем используется в цикле `while` для проверки, нужно ли продолжать выполнение цикла, в последнем операторе которого эта переменная увеличивается на единицу. Такие циклы можно компактно выразить в Java с помощью записи `for`:

```
for (<инициализация>; <логическое выражение>; <увеличение>)
{
   <операторы блока>
}
```

Этот код, лишь с несколькими исключениями, эквивалентен коду

```
<инициализация>;
while (<логическое выражение>)
{
   <операторы блока>
   <увеличение>;
}
```

Циклы `for` предназначены для поддержки этой идиомы программирования — инициализация и увеличение (табл. 1.1.3).

Таблица 1.1.3. Операторы Java

Оператор	Примеры	Определение
Объявление	`int i;` `double c;`	Создание переменной указанного типа с указанным идентификатором
Присваивание	`a = b + 3;` `discriminant = b*b - 4.0*c;`	Присваивание значения типа данных переменной
Инициализирующее объявление	`int i = 1;` `double c = 3.141592625;`	Объявление с одновременным присваиванием начального значения
Неявное присваивание	`i++;` `i += 1;`	`i = i + 1;`
Условный оператор (`if`)	`if (x < 0) x = -x;`	Выполнение оператора в зависимости от логического значения
Условный оператор (`if-else`)	`if (x > y) max = x;` `else       max = y;`	Выполнение одного или другого оператора в зависимости от логического значения
Цикл (`while`)	**`int v = 0;`** `while (v <= N)` `  v = 2*v;` **`double t = c;`** `while (Math.abs(t - c/t) > 1e-15*t)` `  t = (c/t + t) / 2.0;`	Выполнение оператора, пока логическое выражение не станет равным `false`
Цикл (`for`)	`for (int i = 1; i <= N; i++)` `  sum += 1.0/i;` `for (int i = 0; i <= N; i++)` `  StdOut.println(2*Math.PI*i/N);`	Компактная версия оператора `while`
Вызов	`int key = StdIn.readInt();`	Вызов других методов
Возврат	`return false;`	Возврат из метода

Массивы

Массив хранит последовательность значений, которые имеют один и тот же тип. Необходимо не только хранить значения, но и иметь доступ к каждому из них. Для ссылки на отдельные значения массива применяется нумерация, а затем *индексирование*. Если имеется N значений, то считается, что они пронумерованы от 0 до $N-1$. После этого можно однозначно указать в Java-коде любое из них. Для обращения к `i`-тому значению применяется обозначение `a[i]`, для любого значения `i` от `0` до `N-1`. Такая конструкция называется *одномерным массивом*.

2 Зак. 3239

Создание и инициализация массива

Чтобы создать массив в Java-программе, нужно выполнить три отдельных шага:

- объявление имени и типа массива;
- создание массива;
- инициализация значений массива.

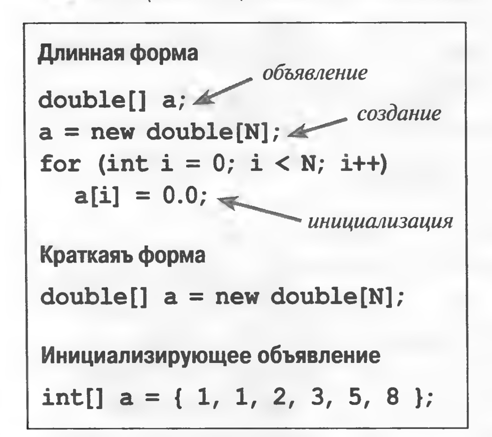


Рис. 1.1.2. Объявление, создание и инициализация массива

Для объявления массива необходимо указать имя и тип данных, которые он будет содержать. Для создания нужно указать его длину (количество значений).

Например, "длинная форма" кода, приведенная на рис. 1.1.2, создает массив из N чисел типа double, каждое из которых инициализируется значением 0.0. Первый оператор — это объявление массива. Оно похоже на объявление обычной переменной соответствующего примитивного типа, но за именем типа находятся квадратные скобки, что и означает, что объявляется именно массив. Ключевое слово new во втором операторе — это директива Java для создания массива. Явное создание массивов во время выполнения необходимо потому, что компилятор Java не может знать на этапе компиляции, какой объем памяти необходим массиву (в отличие от переменных примитивных типов). Оператор for инициализирует N значений массива. Этот код заносит во все элементы массива значение 0.0. При написании кода с использованием массива необходимо быть уверенным, что в коде выполняются объявление, создание и инициализация этого массива. Пропуск одного из этих шагов — распространенная программистская ошибка.

Краткая форма

Для сокращения кода мы часто будем использовать стандартное соглашение Java об инициализации и объединять все три шага в единый оператор, как в "краткой форме" на рис. 1.1.2. Код слева от знака присваивания составляет объявление, а код справа выполняет создание. В данном случае цикл for не нужен, т.к. стандартное значение переменных типа double в массивах языка Java равно 0.0. Однако он может понадобиться, если требуется ненулевое начальное значение. Для числовых типов стандартное начальное значение равно нулю, а для логических — false. Третий вариант, приведенный на рисунке, позволяет указать начальные значения во время компиляции: они перечисляются в виде литералов между фигурными скобками и разделяются запятыми.

Применение массивов

Типичный код обработки массива приведен в табл. 1.1.4. После объявления и создания массива можно обращаться к любому отдельному его значению в любом месте программы, где может находиться имя переменной. Для этого после имени массива нужно дописать целочисленный индекс в квадратных скобках. После создания массива его размер остается постоянным. Программа может узнать длину массива a[] с помощью выражения a.length. Последний элемент массива a[] — это всегда a[a.length-1].

Java выполняет *автоматическую проверку границ*: если для доступа к массиву размером `N` используется индекс, значение которого меньше `0` или больше `N-1`, программа аварийно завершится с исключением времени выполнения `ArrayOutOfBoundsException`.

Таблица 1.1.4. Типичные фрагменты кода для обработки массивов

Задача	Реализация (фрагмент кода)
Поиск максимального значения в массиве	`double max = a[0];` `for (int i = 1; i < a.length; i++)` `   if (a[i] > max) max = a[i];`
Вычисление среднего значения массива	`int N = a.length;` `double sum = 0.0;` `for (int i = 0; i < N; i++)` `   sum += a[i];` `double average = sum / N;`
Копирование значений в другой массив	`int N = a.length;` `double[] b = new double[N];` `for (int i = 0; i < N; i++)` `   b[i] = a[i];`
Перестановка элементов массива в обратном порядке	`int N = a.length;` `for (int i = 0; i < N/2; i++)` `{` `   double temp = a[i];` `   a[i] = a[N-1-i];` `   a[N-i-1] = temp;` `}`
Умножение матрицы на матрицу (квадратные матрицы) **`a[][]*b[][] = c[][]`**	`int N = a.length;` `double[][] c = new double[N][N];` `for (int i = 0; i < N; i++)` `   for (int j = 0; j < N; j++)` `   { // Вычисление скалярного произведения` `     // строки i и столбца j.` `      for (int k = 0; k < N; k++)` `         c[i][j] += a[i][k]*b[k][j];` `   }`

Наложение

Накрепко запомните, что *имя массива относится ко всему массиву*: если присвоить имя одного массива другому, то оба имени будут указывать на один и тот же массив, как показано в следующем фрагменте кода:

```
int[] a = new int[N];
...
a[i] = 1234;
...
int[] b = a;
...
b[i] = 5678;  // Теперь a[i] равно 5678.
```

Эта ситуация называется *наложением* и может приводить к трудноуловимым ошибкам. Если требуется создать копию массива, то нужно объявить, создать и инициализировать новый массив, а затем скопировать все элементы исходного массива в новый массив — как в третьем примере в табл. 1.1.4.

Двумерные массивы

Двумерный массив в Java — это массив одномерных массивов. Двумерный массив может быть *ступенчатым*, когда одномерные массивы имеют разные длины, но чаще всего приходится работать с двумерными массивами размером $M \times N$ — т.е. с массивами из *M строк*, каждая из которых представляет собой массив длиной *N* (поэтому можно говорить, что массив состоит из *N столбцов*). Расширение конструкций массивов в Java на двумерный случай выполняется вполне естественно. Для обращения к элементу в строке `i` и столбце `j` двумерного массива `a[][]` используется запись `a[i][j]`; для объявления двумерного массива нужно добавить еще одну пару квадратных скобок; для создания такого массива после имени типа указывается количество строк, а затем количество столбцов (оба в квадратных скобках):

```
double[][] a = new double[M][N];
```

Такой массив называется массивом $M \times N$. По соглашению первый размер означает количество строк, а второй — количество столбцов. Как и в случае одномерных массивов, Java инициализирует все элементы массивов числовых типов нулями, а логических типов — значениями `false`. Инициализация по умолчанию двумерных массивов скрывает больше кода, чем в случае одномерных массивов. Рассмотренной выше однострочной идиоме создания и инициализации эквивалентен следующий код:

```
double[][] a;
a = new double[M][N];
for (int i = 0; i < M; i++)
   for (int j = 0; j < N; j++)
      a[i][j] = 0.0;
```

Этот код не обязателен для инициализации нулями, но для инициализации другими значениями без вложенных циклов не обойтись.

Статические методы

Каждая Java-программа в данной книге представляет собой либо *определение типа данных* (об этом рассказывается в разделе 1.2), либо *библиотеку статических методов* (которые будут описаны здесь). Во многих языках программирования статические методы называются *функциями*, т.к. они действительно ведут себя как математические функции. Каждый статический метод — это последовательность операторов, которые при *вызове* статического метода выполняются один за другим так, как описано ниже. Слово *статические* отличает эти методы от *методов экземпляров*, которые будут рассмотрены в разделе 1.2. При описании характеристик, присущих обоим видам методов, мы будем употреблять слово *метод* без уточнения.

Определение статического метода

Метод инкапсулирует вычисление, которое определено как последовательность операторов. Метод принимает *аргументы* (значения заданных типов данных) и на основе этих аргументов вычисляет *возвращаемое значение* какого-то типа данных (вроде значения, определенного математической функцией) или вызывает *побочный эффект*, который зависит от аргументов (например, вывод значения). Статический метод `rank()` в классе `BinarySearch` является примером первого вида, а метод `main()` — второго.

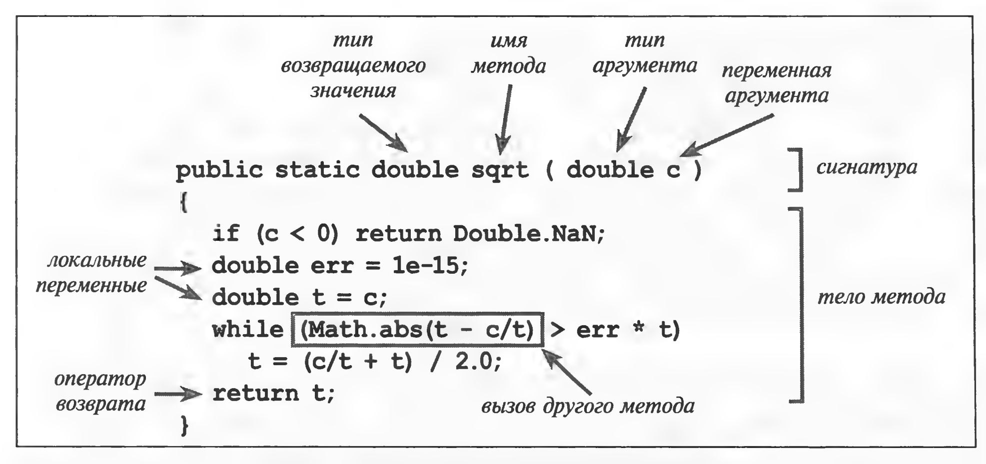


Рис. 1.1.3. Структура статического метода

Каждый статический метод состоит (рис. 1.1.3) из *сигнатуры* (ключевые слова `public static`, за которыми следует тип возвращаемого значения, имя метода и последовательность аргументов, каждый с объявленным типом) и *тела* (блок операторов — последовательность операторов, заключенная в фигурные скобки). Примеры статических методов приведены в табл. 1.1.5.

Таблица 1.1.5. Типичные реализации статических методов

Задача	Реализация
Абсолютная величина значения **int**	`public static int abs(int x)` `{` `   if (x < 0) return -x;` `   else      return  x;` `}`
Абсолютная величина значения **double**	`public static double abs(double x)` `{` `   if (x < 0.0) return -x;` `   else        return  x;` `}`
Проверка, простое ли число	`public static boolean isPrime(int N)` `{` `   if (N < 2) return false;` `   for (int i = 2; i*i <= N; i++)` `      if (N % i == 0) return false;` `   return true;` `}`
Квадратный корень (метод Ньютона)	`public static double sqrt(double c)` `{` `   if (c < 0) return Double.NaN;` `   double err = 1e-15;` `   double t = c;` `   while (Math.abs(t - c/t) > err * t)` `      t = (c/t + t) / 2.0;`

Окончание табл. 1.1.5

Задача	Реализация
Гипотенуза прямоугольного треугольника	`public static double hypotenuse( double a, double b) { return Math.sqrt(a*a + b*b); }`
Гармоническое число	`public static double H(int N) { double sum = 0.0; for (int i = 1; i <= N; i++) sum += 1.0 / i; return sum; }`

Вызов статического метода

Вызов статического метода — это его имя, за которым в скобках следуют выражения, задающие значения аргументов и разделяемые запятыми. Если вызов метода является частью выражения, то метод вычисляет значение, которое затем подставляется в выражение на место вызова. Например, вызов `rank()` в методе `BinarySearch()` возвращает значение типа `int`. Вызов метода, за которым следует точка с запятой, представляет собой *оператор*, который обычно выполняет побочные эффекты. Например, вызов `Arrays.sort()` в методе `main()` в классе `BinarySearch` обращается к системному методу `Arrays.sort()`, который в качестве побочного эффекта упорядочивает элементы массива. При вызове метода его переменные аргументов инициализируются значениями соответствующих выражений из вызова. Оператор `return` завершает выполнение статического метода и возвращает управление вызвавшему методу. Если статический метод предназначен для вычисления значения, это значение должно быть указано в операторе `return`. Если статический метод может дойти до конца последовательности своих операторов, не выполнив оператор `return`, компилятор расценит это как ошибку.

Свойства методов

Полное и подробное описание свойств методов выходит за рамки нашего изложения, но некоторые моменты следует упомянуть.

- *Аргументы передаются по значению.* Переменные аргументов можно использовать в любом месте тела метода так же, как и локальные переменные. Единственное различие между ними состоит в том, что переменная аргумента инициализируется значением аргумента, взятым из кода вызова. Метод работает со значениями своих аргументов, но не с самими аргументами. Одно из следствий этого подхода: изменение значения переменной аргумента в теле статического метода не влияет на вызвавший код. В данной книге мы, как правило, не изменяем переменные аргументов. Соглашение о передаче по значению означает, что для аргументов-массивов выполняется наложение (см. выше раздел “Наложение”): метод использует переменную аргумента для обращения к массиву вызвавшего метода и может изменить его содержимое (хотя не может изменить сам массив). Например, вызов `Arrays.sort()`, конечно, изменяет содержимое массива, переданного в качестве аргумента — он упорядочивает его элементы.
- *Имена методов могут быть перегружены.* Например, этот подход используется в Java-библиотеке `Math`, чтобы предоставить реализации для всех примитивных чи-

словых типов. Перегрузка также часто применяется для определения двух различных версий функции, одна из которых принимает аргумент, а другая использует стандартное значение этого аргумента.

- *Метод возвращает единственное значение, но может содержать несколько операторов возврата.* Java-метод может возвратить только одно значение — того типа, который объявлен в сигнатуре метода. Управление возвращается в вызывающую программу, как только в статическом методе достигается первый оператор `return`. Операторы `return` можно размещать где угодно. Но даже при наличии нескольких операторов `return` любой статический метод возвращает единственное значение при любом его вызове — это значение, записанное после первого же выполненного оператора `return`.
- *Метод может иметь побочные эффекты.* В качестве возвращаемого типа метод может использовать ключевое слово `void`, что означает отсутствие возвращаемого значения. В таком статическом методе явный оператор возврата не обязателен: управление передается вызывающему методу после выполнения последнего оператора. Говорят, что статический метод `void` порождает побочные эффекты — использует входные данные, генерирует выходные данные, изменяет элементы в массиве или как-то еще изменяет состояние системы. К примеру, статический метод `main()` в наших программах имеет возвращаемый тип `void`, т.к. он предназначен для формирования выходных данных. Технически методы `void` не реализуют математические функции (как и метод `Math.random()`, который не принимает аргументы, хотя и генерирует возвращаемое значение).

Методы экземпляров, которые будут рассмотрены в разделе 2.1, также обладают этими свойствами, хотя порождение побочных эффектов выполняется в них существенно иначе.

Рекурсия

Метод может вызвать сам себя — это называется *рекурсией*. (Если вы не очень знакомы с этим принципом, рекомендуем проработать упражнения 1.1.16–1.1.22). Например, код в листинге 1.1.1 содержит альтернативную реализацию метода `rank()` в методе `BinarySearch()`. Мы будем часто пользоваться рекурсивными реализациями методов, поскольку они приводят к компактному и элегантному коду, который легче воспринимается, чем соответствующая реализация без рекурсии. К примеру, комментарий в листинге 1.1.1 содержит краткое описание назначения кода. Данный комментарий можно использовать для доказательства правильности работы метода с помощью математической индукции. Мы еще вернемся к этой теме и представим соответствующее доказательство для бинарного поиска в разделе 3.1. При разработке рекурсивных программ нужно помнить о трех важных правилах.

- В рекурсии должен быть *базовый вариант*: мы всегда помещаем в начало рекурсивной программы условный оператор с оператором `return`.
- Рекурсивные вызовы должны обращаться к подзадачам, *меньшим* в некотором смысле, чтобы рекурсивные вызовы сходились к базовому варианту. В листинге 1.1.1 разность значений четвертого и третьего аргументов постоянно уменьшается.
- Рекурсивные вызовы не должны обращаться к *перекрывающимся* подзадачам. В листинге 1.1.1 части массива, просматриваемые двумя подзадачами, не имеют общих элементов.

Листинг 1.1.1. Рекурсивная реализация бинарного поиска

```
public static int   rank(int key, int[] a)
{  return rank(key, a, 0, a.length - 1);  }
public static int rank(int key, int[] a, int lo, int hi)
{  // Если key присутствует в a[], его индекс не меньше lo и не больше hi.
   if (lo > hi) return -1;
   int mid = lo + (hi - lo) / 2;
   if      (key < a[mid]) return rank(key, a, lo, mid - 1);
   else if (key > a[mid]) return rank(key, a, mid + 1, hi);
   else                   return mid;
}
```

Несоблюдение данных правил может привести к неверным результатами или к чрезвычайно неэффективной программе (см. упражнения 1.1.19 и 1.1.27). Эти правила предназначены для получения ясных и корректных программ, производительность которых легко оценить. Еще одна причина использования рекурсивных методов — они приводят к математическим моделям, которые позволяют оценить производительность. Этот вопрос будет рассмотрен при изучении бинарного поиска в разделе 3.2 и в ряде других мест книги.

Базовая модель программирования

Библиотека статических методов — это набор статических методов, которые определены в Java-классе с помощью создания специального файла. Этот файл должен содержать ключевые слова `public class`, за ними имя класса, а за ним статические методы в фигурных скобках; он должен быть преобразован в файл с таким же именем, как у класса, но с расширением `.java`. Базовая модель для программирования на Java — это разработка программы, предназначенной для решения конкретной вычислительной задачи, с помощью создания библиотеки статических методов, один из которых называется `main()`. Ввод в командной строке команды `java` с последующим именем класса, за которым идет последовательность строк, приводит к вызову метода `main()` из этого класса с аргументом, который представляет собой массив с этими строками. После выполнения последнего оператора из метода `main()` программа прекращает работу. Когда в данной книге мы говорим о *Java-программе* для выполнения какой-то задачи, мы имеем в виду код, разработанный по описанному принципу (и, возможно, содержащий определение типа данных, как описано в разделе 1.2). Например, `BinarySearch` — это Java-программа, состоящая из двух статических методов `rank()` и `main()` и выполняющая задачу вывода в выходной поток чисел, которые отсутствуют в файле белого списка, указанном в аргументе командной строки.

Модульное программирование

Очень важным в этой модели является то, что библиотеки статических методов делают возможным *модульное программирование* — когда создаются библиотеки статических методов (*модули*), и статический метод из одной библиотеки может вызывать статические методы, определенные в других библиотеках. Этот подход имеет множество важных преимуществ. Он позволяет:

- работать с модулями обозримого размера, даже в программах, содержащих большой объем кода;

- совместно и повторно использовать код без необходимости реализовывать его снова;
- легко заменять реализации улучшенными версиями;
- разрабатывать подходящие абстрактные модели для решения программных задач;
- локализовать поиск ошибок (см. ниже абзац о тестировании модулей).

Например, в классе `BinarySearch` используются три независимо разработанные библиотеки: наши библиотеки `StdIn` и `In` и Java-библиотека `Arrays`. Каждая из этих библиотек, в свою очередь, обращается к нескольким другим библиотекам.

Тестирование модулей

При программировании на Java рекомендуется включать в каждую библиотеку статических методов метод `main()`, предназначенный для тестирования методов из этой библиотеки (ряд других языков программирования не позволяют иметь несколько методов `main()` и поэтому не поддерживают такой подход). Надлежащее тестирование модуля само по себе может оказаться серьезной программистской задачей. Как минимум, каждый модуль должен содержать метод `main()`, который выполняет код модуля и как-то демонстрирует его работоспособность. При развитии модуля метод `main()` зачастую также изменяется и становится или *клиентом разработки*, который помогает выполнить более тщательные проверки при разработке кода, или *клиентом тестирования*, который помогает выполнить тщательное тестирование всего кода. Из-за усложнения клиента его можно вынести в независимый модуль. В данной книге мы используем метод `main()` для демонстрации назначения каждого модуля, а клиенты тестирования оставляем читателям в качестве упражнений.

Внешние библиотеки

Мы применяем статические методы для создания четырех различных видов библиотек, каждая из которых требует (слегка) различных процедур для использования ее кода. В основном это библиотеки статических методов, но некоторые из них — это определения типов данных, которые содержат и некоторые статические методы.

- Стандартные системные библиотеки `java.lang.*`. Это следующие библиотеки: `Math` с методами для часто используемых математических функций; `Integer` и `Double` для преобразований между символьными строками, целыми и вещественными значениями; `String` и `StringBuilder`, которые будут подробно рассмотрены ниже в данном разделе и в главе 5; а также десятки других библиотек, которые нам не пригодятся.
- Импортированные системные библиотеки вроде `java.util.Arrays`. В любом стандартном выпуске Java существуют тысячи таких библиотек, но мы почти не будем применять их в данной книге. Для их использования необходим оператор `import` в начале программы.
- Другие библиотеки из данной книги. Например, метод `rank()` из класса `BinarySearch` может быть задействован в другой программе. Для этого потребуется загрузить исходный код с сайта книги в рабочий каталог.
- Стандартные библиотеки `Std*`, которые мы разработали для целей данной книги. Эти библиотеки будут кратко описаны на следующих нескольких страницах. Исходный код и инструкции по загрузке доступны на сайте книги.

Стандартные системные библиотеки

Math
Integer†
Double†
String†
StringBuilder
System

Импортированные системные библиотеки

java.util.Arrays

Наши стандартные библиотеки

StdIn
StdOut
StdDraw
StdRandom
StdStats
In†
Out†

† *определения типов данных, которые содержат некоторые статические методы*

Рис. 1.1.4. *Библиотеки со статическими методами, используемые в настоящей книге*

Для вызова метода, находящегося в другой библиотеке (в том же каталоге или в заданном каталоге, в стандартной системной библиотеке или системной библиотеке, которая указана в операторе `import` перед определением класса), необходимо в каждом вызове перед именем метода добавлять имя библиотеки с последующей точкой. Например, метод `main()` из класса `BinarySearch` вызывает метод `sort()` из системной библиотеки `java.util.Arrays`, метод `readInts()` из нашей библиотеки `In` и метод `println()` из нашей библиотеки `StdOut`.

Библиотеки методов, реализованные самостоятельно или кем-то еще, в среде модульного программирования могут значительно расширить область применения нашей модели программирования. Кроме всех библиотек, доступных в стандартном выпуске Java, в сети можно найти тысячи других, предназначенных для всевозможных целей. Чтобы ограничить область действия нашей модели программирования до обозримого размера и не отвлекаться от самих алгоритмов, мы задействуем только библиотеки, перечисленные на рис. 1.1.4, и подмножество их методов, перечисленных в *API-интерфейсах*, о которых сейчас пойдет речь.

API-иитерфейсы

Критически важным компонентом модульного программирования является *документация*, которая объясняет функционирование методов библиотеки, предназначенных для использования другими. Мы будем единообразно описывать необходимые нам методы библиотек в *интерфейсах прикладного программирования (API-интерфейсах)*, которые содержат имя библиотеки и сигнатуры и краткие описания всех задействованных нами методов. Мы называем *клиентом* программу, которая вызывает какой-то метод из другой библиотеки, и *реализацией* — Java-код, который реализует методы из API-интерфейса.

Пример

На рис. 1.1.5 приведен пример API-интерфейса для часто используемых статических методов из стандартной библиотеки `Math` из пакета `java.lang` — просто для демонстрации наших соглашений по API-интерфейсам. Эти методы реализуют математические функции: на основе значения своего аргумента они вычисляют значение указанного типа (кроме `random()` — этот метод не реализует математическую функцию, т.к. не принимает аргумент). Все они работают со значениями типа `double` и возвращают результат этого же типа, поэтому их можно рассматривать как расширение типа `double`; подобная расширяемость является одной из характерных черт современных языков программирования. Каждому методу соответствует строка в API-интерфейсе, содержащая информацию, которая необходима для использования этого метода.

public class **Math**	
`static double abs(double a)`	*абсолютное значение* `a`
`static double max(double a, double b)`	*максимальное значение из* `a` *и* `b`
`static double min(double a, double b)`	*минимальное значение из* `a` *и* `b`
Примечание 1. Функции `abs()`, `max()` *и* `min()` *определены также для* `int`, `long` *и* `float`.	
`static double sin(double theta)`	*функция синуса*
`static double cos(double theta)`	*функция косинуса*
`static double tan(double theta)`	*функция тангенса*
Примечание 2. Углы выражаются в радианах. Для преобразования в градусы и обратно используйте методы `toDegrees()` *и* `toRadians()`.	
Примечание 3. Для обратных функций имеются функции `asin()`, `acos()` *и* `atan()`.	
`static double exp(double a)`	*экспонента* (e^a)
`static double log(double a)`	*натуральный логарифм* ($\log_e a$ *или* $\ln a$)
`static double pow(double a, double b)`	*возведение a в степень b* (a^b)
`static double random()`	*случайное число из диапазона* [0, 1)
`static double sqrt(double a)`	*квадратный корень из* `a`
`static double E`	*значение e (константа)*
`static double PI`	*значение π (константа)*
Другие доступные функции описаны на сайте книги.	

***Рис. 1.1.5.** API-интерфейс для математической библиотеки Java (фрагмент)*

Библиотека `Math` определяет также точные значения констант `PI` (для π) и `E` (для *e*), которые позволяют использовать эти имена вместо самих констант. Например, значение `Math.sin(Math.PI/2)` равно `1.0`, и значение `Math.log(Math.E)` тоже равно `1.0` — поскольку метод `Math.sin()` принимает аргумент, выраженный в радианах, а метод `Math.log()` реализует функцию натурального логарифма.

Java-библиотеки

В состав каждого выпуска Java входят онлайновые описания тысяч библиотек, но мы возьмем из них лишь несколько методов, которые будут задействованы в данной книге — чтобы четко обрисовать нашу модель программирования. Например, в классе `BinarySearch` используется метод `sort()` из Java-библиотеки `Arrays`, описание которой приведено на рис. 1.1.6.

public class **Arrays**	
`static void sort(int[] a)`	*упорядочение массива по возрастанию*
Примечание. Этот метод определен и для других примитивных типов, а также для `Object`.	

***Рис. 1.1.6.** Выдержка из Java-библиотеки* `Arrays (java.util.Arrays)`

Библиотека Arrays не входит в пакет java.lang, поэтому для ее использования необходим оператор import, как в примере BinarySearch. Глава 2 посвящена реализациям метода sort() для массивов, в том числе и алгоритмам сортировки слиянием и быстрой сортировки, которые реализованы в методе Arrays.sort(). Многие фундаментальные алгоритмы, рассматриваемые в данной книге, реализованы как на Java, так и во многих других средах программирования. К примеру, библиотека Arrays содержит и реализацию бинарной сортировки. Чтобы не было путаницы, мы обычно используем наши собственные реализации, хотя нет ничего плохого в применении отточенных библиотечных реализаций алгоритма, в котором вы четко разобрались.

Наши стандартные библиотеки

Мы разработали ряд библиотек, которые предоставляют полезные возможности для обучения программированию на Java, для научных приложений и для разработки, изучения и применения алгоритмов. В основном эти библиотеки предназначены для организации ввода и вывода; но имеются еще две библиотеки для тестирования и анализа наших реализаций. Первая из них (рис. 1.1.7) расширяет метод Math.random(), чтобы иметь возможность генерировать случайные значения для различных распределений, а вторая (рис. 1.1.8) поддерживает статистические вычисления.

public class **StdRandom**			
static	void	setSeed(long seed)	*инициализация*
static	double	random()	*вещественное от* 0 *до* 1
static	int	uniform(int N)	*целое от* 0 *до* N-1
static	int	uniform(int lo, int hi)	*целое от* lo *до* hi-1
static	double	uniform(double lo, double hi)	*вещественное от* lo *до* hi
static	boolean	bernoulli(double p)	true *с вероятностью* p
static	double	gaussian()	*нормальное со средним* 0 *и ср.-кв. откл.* 1
static	double	gaussian(double m, double s)	*нормальное со средним* m *и ср.-кв. откл.* s
static	int	discrete(double[] a)	i *с вероятностью* a[i]
static	void	shuffle(double[] a)	*случайное перемешивание массива* a[]

Примечание. Перегруженные реализации shuffle() *имеются и для других примитивных типов, а также для* Object.

Рис. 1.1.7. *API-интерфейс для нашей библиотеки статических методов генерации случайных чисел*

public class **StdStats**	
static double max(double[] a)	*максимальное значение*
static double min(double[] a)	*минимальное значение*
static double mean(double[] a)	*среднее значение*
static double var(double[] a)	*дисперсия выборки*
static double stddev(double[] a)	*среднеквадратичное отклонение выборки*
static double median(double[] a)	*медиана*

Рис. 1.1.8. *API-интерфейс для нашей библиотеки статических методов анализа данных*

Метод `initialize()` из класса `StdRandom` позволяет выполнить *инициализацию* генератора случайных чисел, чтобы иметь возможность повторять эксперименты с использованием случайных чисел. Реализации многих из этих методов приведены для справки в табл. 1.1.6. Некоторые из них реализовать чрезвычайно легко — и зачем тогда оформлять их в виде библиотеки? Ответы на этот вопрос стандартны для хорошо спроектированных библиотек.

- В них реализован уровень абстракции, который позволяет сконцентрироваться на реализации и тестировании алгоритмов, а не на генерировании случайных объектов или подсчете статистических данных. Клиентский код, использующий такие методы, яснее и проще для понимания, чем доморощенный код, который выполняет те же вычисления.
- Библиотечные реализации содержат проверки на исключительные условия, охватывают редкие ситуации и проходят тщательную проверку, поэтому можно быть вполне уверенным, что они будут работать как положено. Такие реализации могут содержать значительный объем кода. Нам часто бывают нужны реализации для различных типов данных. Например, Java-библиотека `Arrays` содержит несколько перегруженных реализаций метода `sort()` — для каждого типа данных, который может понадобиться сортировать.

Таблица 1.1.6. Реализации статических методов в библиотеке `StdRandom`

Требуемый результат	Реализация
Случайное значение `double` из диапазона `[a,b)`	`public static double uniform(double a, double b)` `{ return a + StdRandom.random() * (b-a); }`
Случайное значение `int` из диапазона `[0..N)`	`public static int uniform(int N)` `{ return (int) (StdRandom.random() * N); }`
Случайное значение `int` из диапазона `[lo..hi)`	`public static int uniform(int lo, int hi)` `{ return lo + StdRandom.uniform(hi - lo); }`
Случайное значение `int` из дискретного распределения (`i` с вероятностью `a[i]`)	`public static int discrete(double[] a)` `{ // Сумма элементов a[] должна быть равна 1.` `double r = StdRandom.random();` `double sum = 0.0;` `for (int i = 0; i < a.length; i++)` `{` `sum = sum + a[i];` `if (sum >= r) return i;` `}` `return -1;` `}`
Случайное перемешивание элементов в массиве значений `double` (см. упражнение 1.1.36)	`public static void shuffle(double[] a)` `{` `int N = a.length;` `for (int i = 0; i < N; i++)` `{ //Обмен a[i] со случайным элементом из a[i..N-1].` `int r = i + StdRandom.uniform(N-i);` `double temp = a[i];` `a[i] = a[r];` `a[r] = temp;` `}` `}`

Это основные доводы в пользу модульного программирования в Java, хотя в данном случае они, возможно, не так важны. Методы в обеих этих библиотеках в достаточной степени самодокументированны, и многие из них нетрудно реализовать, хотя некоторые все же представляют собой интересные алгоритмические упражнения. Поэтому мы советуем *как* изучить код `StdRandom.java` и `StdStats.java`, представленный на сайте книги, *так и* пользоваться этими проверенными реализациями. Для их применения (и знакомства с кодом) проще всего загрузить их код с сайта книги в свой рабочий каталог. Различные системные механизмы для их использования без создания нескольких копий также описаны на сайте книги.

Ваши собственные библиотеки

Неплохо рассматривать *каждую написанную вами программу* как библиотечную реализацию, которая может понадобиться в дальнейшем для каких-то целей.

- Напишите код для клиента — реализации самого высокого уровня, которая разбивает вычисление на обозримые части.
- Сформулируйте API-интерфейс для библиотеки (или несколько API-интерфейсов для нескольких библиотек) статических методов, в которой описаны все части.
- Разработайте реализацию API-интерфейса с методом `main()`, который проверяет работу методов независимо от клиента.

Этот подход не только позволит обзавестись ценным ПО, которое может пригодиться в дальнейшем: подобное использование модульного программирования — ключ для успешной работы над сложными программистскими задачами.

API-интерфейс предназначен для *отделения* клиента от реализации: клиент ничего не должен знать о реализации, кроме информации, приведенной в API-интерфейсе, а реализация не должна учитывать особенности конкретного клиента. API-интерфейсы позволяют отдельно разработать код для произвольных целей, а затем использовать его сколько угодно. Ни одна Java-библиотека может не содержать все методы, необходимые для конкретного вычисления, поэтому эта возможность является важным шагом для разработки сложных программных приложений. В связи с этим программисты обычно рассматривают API-интерфейс как *соглашение* между клиентом и реализацией, т.е. ясное описание того, что должен делать каждый метод. Наша цель при реализации — соблюдать условия этого соглашения. Часто это можно сделать различными способами, и отделение клиентского кода от кода реализации позволяет поставлять новые и усовершенствованные реализации. При изучении алгоритмов эта возможность является важным средством выяснить влияние разрабатываемых алгоритмических усовершенствований.

Строки

Строка (`String`) представляет собой последовательность символов (значений типа `char`). Литеральная строка — это последовательность символов, заключенная в двойные кавычки, например, `"Hello, World"`. Тип данных `String` является типом данных языка Java, но это *не* примитивный тип. Мы рассматриваем его сейчас потому, что это фундаментальный тип данных, который используется практически во всех Java-программах.

Конкатенация

В Java имеется встроенная операция *конкатенации* (+) для типа `String`, которая похожа на встроенные операции для примитивных типов, и поэтому строку, приведенную в табл. 1.1.7, можно было бы добавить к табл. 1.1.2. Результатом конкатенации двух значений типа `String` является одно значение типа `String`: первая строка, к которой добавлена вторая строка.

Таблица 1.1.7. Тип данных `String` в Java

Тип	Множество значений	Типичные литералы	Операции	Типичные выражения: выражение	значение
`String`	Последовательности символов	`"AB"`	+ (конкатенация)	`"Hi, " + "Bob"`	`"Hi, Bob"`
		`"Hello"`		`"12" + "34"`	`"1234"`
		`"2.5"`		`"1" + "+" + "2"`	`"1+2"`

Преобразование

Два основных назначения строк — это преобразование значений, введенных с клавиатуры, в значения типов данных и преобразование значений типов данных в значения, которые можно прочитать на дисплее. В Java имеются встроенные операции для типа `String`, которые позволяют выполнять такие операции. В частности, язык содержит библиотеки `Integer` и `Double`, в которых имеются статические методы для преобразования значений `String` в значения `int` и обратно, а также для преобразования значений `String` в значения `double` и обратно (рис. 1.1.9).

```
public class Integer
static      int parseInt(String s)        преобразование s в значение int
static   String toString(int i)           преобразование i в значение String

public class Double
static   double parseDouble(String s)     преобразование s в значение double
static   String toString(double x)        преобразование x в значение String
```

***Рис. 1.1.9.** API-интерфейсы для преобразования чисел в строки и обратно*

Автоматическое преобразование

Мы нечасто будем пользоваться описанными на рис. 1.1.9 статическими методами `toString()`, т.к. в Java есть встроенный механизм, позволяющий преобразовать значение произвольного типа данных в значение `String` с помощью конкатенации: если *один* из аргументов операции + является строкой, Java *автоматически* преобразует другой аргумент в тип `String` (если он другого типа). Кроме применения вроде `"Квадратный корень из 2.0 равен " + Math.sqrt(2.0)`, этот механизм позволяет преобразовать в строку любое значение, конкатенировав его с пустой строкой `""`.

Аргументы командной строки

Одно из важных способов применения строк в программировании на Java — это механизм передачи информации из командной строки в программу. Этот механизм работает просто. Если напечатать в окне командной строки команду `java`, а за ней имя библиотеки и еще последовательность строк, то система Java вызывает метод `main()` из этой библиотеки с *массивом строк* в качестве аргумента; этот массив содержит строки, введенные после имени библиотеки. Например, метод `main()` из класса `BinarySearch` принимает из командной строки один аргумент, поэтому система создает массив с одним элементом. Программа использует это значение (`args[0]`) в качестве аргумента метода `In.readInts()` — как имя файла, содержащего белый список. Часто бывает нужно, чтобы аргумент командной строки представлял собой число, и тогда понадобится метод `parseInt()` для преобразования в значение `int` или `parseDouble()` для преобразования в значение `double`.

Обработка строк — важный компонент современной компьютерной обработки данных. Пока мы будем использовать тип `String` только для преобразования чисел во внешнем представлении (последовательность символов) во внутреннее представление значений числовых типов данных и обратно. В разделе 1.2 мы узнаем, что Java поддерживает гораздо больше операций над значениями `String`, которые мы также будем использовать в данной книге; в разделе 1.4 мы познакомимся с внутренним представлением значений `String`; а в главе 5 мы подробно рассмотрим алгоритмы обработки данных типа `String`. Это одни из самых интересных, запутанных и важных методов, которые будут рассматриваться в данной книге.

Ввод и вывод

Основное назначение наших стандартных библиотек для ввода, вывода и рисования — поддержка простой модели для Java-программ, которая позволяет взаимодействовать с внешним миром. Эти библиотеки задействуют мощные возможности, имеющиеся в Java-библиотеках, но обычно гораздо более сложны и трудны для понимания и использования. Сейчас мы вкратце ознакомимся с этой моделью (рис. 1.1.10).

В нашей модели Java-программа принимает входные значения из *аргументов командной строки* либо из абстрактного потока символов, который называется *потоком стандартного ввода*, и записывает значения в другой абстрактный поток символов, который называется *потоком стандартного вывода*.

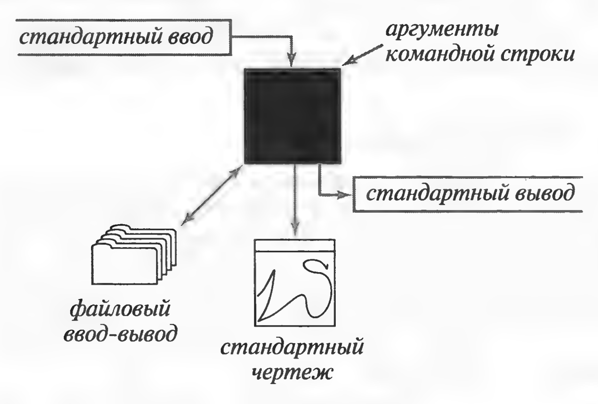


Рис. 1.1.10. *Укрупненное представление Java-программы*

Поэтому нам придется кратко рассмотреть интерфейс между Java и операционной системой, т.е. механизмы, которые имеются в большинстве современных операционных систем и сред разработки ПО. Подробную информацию о вашей конкретной системе можно прочитать на сайте книги. По умолчанию аргументы командной строки, стандартный ввод и стандартный вывод связаны с приложением, поддерживаемым либо операционной системой, либо средой разработки ПО, которая принимает команды. Окно, используемое таким приложением, в котором можно вводить и выводить текст, мы будем обобщенно называть *окном терминала*. Со времен первых Unix-систем в 1970-х годах эта модель зарекомендовала себя как удобный способ для взаимодействия с программами и данными. Мы добавили в классическую модель еще *стандартный чертеж*, который позволяет создавать визуальные представления для анализа данных.

Команды и аргументы

В окне терминала имеется строка приглашения, где можно вводить *команды* для операционной системы, которые могут принимать *аргументы*. В данной книге мы будем применять лишь несколько команд, которые перечислены в табл. 1.1.8. Чаще всего мы будем использовать команду `java`, которая позволяет запускать наши программы (рис. 1.1.11). Как было сказано в разделе “Аргументы командной строки”, Java-классы содержат статический метод `main()`, который принимает в качестве аргумента массив `args[]` типа `String`. Этот массив представляет собой последовательность введенных в командной строке аргументов, которые переданы операционной системой в Java. По соглашению Java и операционная система обрабатывают аргументы как строки. Если нужно, чтобы аргумент был числом, мы используем такой метод, как `Integer.parseInt()`, чтобы преобразовать его из типа `String` в нужный тип.

Таблица 1.1.8. Типичные команды операционной системы

Команда	Аргументы	Назначение
`javac`	Имя `.java`-файла	Компиляция Java-программы
`java`	Имя `.class`-файла (без расширения) и аргументы командной строки	Выполнение Java-программы
`more`	Любое имя текстового файла	Вывод содержимого файла

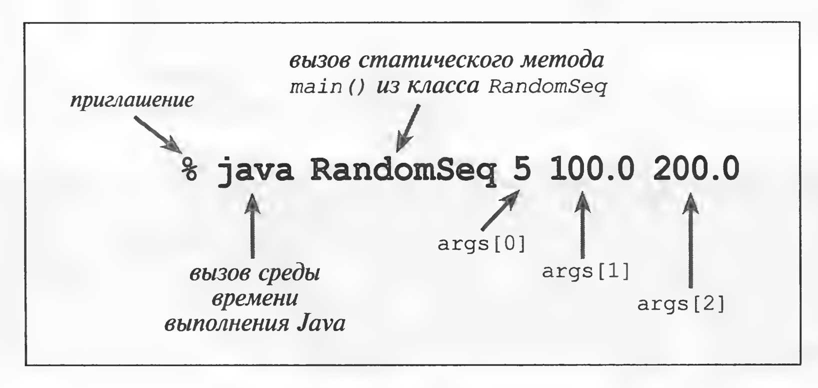


Рис. 1.1.11. Структура команды

Стандартный вывод

Наша библиотека StdOut обеспечивает поддержку стандартного вывода. По умолчанию система подключает стандартный вывод к окну терминала. Метод print() выводит в стандартный вывод свой аргумент; метод println() выводит символ новой строки; а метод printf() обеспечивает форматированный вывод, как будет описано ниже. В Java, в библиотеке System.out, имеется аналогичный метод, но мы будем использовать StdOut, чтобы работать со стандартным вводом и стандартным выводом однотипным образом (с некоторыми техническими усовершенствованиями).

public class **StdOut**	
static void print(String s)	*вывод s*
static void println(String s)	*вывод s с переводом стоки*
static void println()	*перевод строки*
static void printf(String f, ...)	*форматированный вывод*

Рис. 1.1.12. *API-интерфейс статических методов для стандартного вывода*

Чтобы применять эти методы, загрузите с сайта книги в свой рабочий каталог файл StdOut.java и используйте для вызова код наподобие StdOut.println("Hello, World"); Пример клиентского класса приведен в листинге 1.1.2, а пример его работы — на рис. 1.1.13.

Листинг 1.1.2. Пример клиента библиотеки StdOut

```
public class RandomSeq
{
   public static void main(String[] args)
   {  // Вывод N случайных значений из диапазона (lo, hi).
      int N = Integer.parseInt(args[0]);
      double lo = Double.parseDouble(args[1]);
      double hi = Double.parseDouble(args[2]);
      for (int i = 0; i < N; i++)
      {
         double x = StdRandom.uniform(lo, hi);
         StdOut.printf("%.2f\n", x);
      }
   }
}
```

```
% java RandomSeq 5 100.0 200.0
123.43
153.13
144.38
155.18
104.02
```

Рис. 1.1.13. *Пример работы клиентского класса из листинга 1.1.2*

Форматированный вывод

В своей простейшей форме метод printf() принимает два аргумента. Первый аргумент — это *строка формата*, которая описывает, как следует преобразовать второй аргумент в строку для вывода. Самые простые строки формата начинаются с символа % и заканчиваются однобуквенным *кодом преобразования*. Чаще всего мы будем применять коды преобразования d (для десятичных значений из целых типов), f (для значений с плавающей точкой) и s (для строковых значений). Между символом % и кодом преобразования находится целое значение, которое задает *ширину поля* для преобразованного значения (количество символов в преобразованной выводимой строке). По умолчанию для соответствия длины преобразованной выходной строки и заданной ширины поля слева могут добавляться пробелы; если пробелы необходимы справа, перед длиной поля нужно поместить знак минус. (Если преобразованная выходная строка длиннее ширины поля, ширина поля игнорируется.) За шириной поля можно поместить точку, а за ней — количество цифр после десятичной точки (точность) для значений double или количество первых символов из начала строки для значений String (см. табл. 1.1.9). При использовании метода printf() очень важно помнить, что *код преобразования в строке формата и тип соответствующего аргумента должны соответствовать друг другу*. То есть у Java должна быть возможность выполнить преобразование из типа аргумента в тип, требуемый кодом преобразования. Первый аргумент метода printf() имеет тип String и может содержать символы, отличные от строки формата. Такие символы переносятся в вывод без изменений, лишь строка формата заменяется значением аргумента (соответственно преобразованным в тип String). Например, оператор

```
StdOut.printf("ПИ примерно равно %.2f\n", Math.PI);
```

выводит строку

```
ПИ примерно равно 3.14
```

Обратите внимание, что при работе с методом printf() для перевода строки нужно поместить в аргумент символ новой строки \n. Функция printf() может принимать больше двух аргументов. В этом случае строка формата должна содержать спецификаторы формата для каждого дополнительного аргумента — возможно, разделенные другими символами, предназначенными для вывода в выходной поток. Можно также воспользоваться статическим методом String.format(), аргументы которого в точности совпадают с тем, что описано для printf() — так можно получить форматированную строку без ее вывода. Форматированный вывод представляет собой удобный механизм, который позволяет разрабатывать компактный код для получения табулированных экспериментальных данных (основное назначение в данной книге).

Таблица 1.1.9. Преобразование формата для метода printf() (многие другие возможности описаны на сайте книги)

Тип	Код	Типичный литерал	Пример строки формата	Преобразованные строковые значения для вывода
int	d	512	"%14d"	" 512"
			"%-14d"	"512 "
double	f	1595.1680010754388	"%14.2f"	" 1595.17"
	e		"%.7f"	"1595.1680011"
			"%14.4e"	" 1.5952e+03"
String	s	"Hello, World"	"%14s"	" Hello, World"
			"%-14s"	"Hello, World "
			"%-14.5s"	"Hello "

Стандартный ввод

Наша библиотека StdIn принимает значения из потока стандартного ввода, который может быть пустым, а может содержать последовательность значений, разделенных пробельными символами — пробелами, символами табуляции, новой строки и т.п. По умолчанию система связывает стандартный ввод с окном терминала: все введенные с клавиатуры символы представляют собой поток ввода (с завершающим символом <ctrl-d> или <ctrl-z>, в зависимости от конкретного приложения — см. листинг 1.1.3 и рис. 1.1.14). Каждое значение представляет собой значение типа String или одного из примитивных типов Java. Одна из важных особенностей потока стандартного ввода — программа потребляет значения при их чтении, т.е. после прочтения значения его невозможно вернуть и прочитать еще раз. Это свойство накладывает некоторые ограничения, но отражает физические характеристики многих устройств ввода и упрощает реализацию абстракции. В модели потока ввода статические методы из библиотеки StdIn в основном самодокументированны (описаны своими сигнатурами — см. рис. 1.1.15).

Листинг 1.1.3. Пример клиента библиотеки StdIn

```
public class  Average
{
   public static void main(String[] args)
   {  // Среднее значение чисел из StdIn.
      double sum = 0.0;
      int cnt = 0;
      while (!StdIn.isEmpty())
      {  // Чтение числа и накопление суммы.
         sum += StdIn.readDouble();
         cnt++;
      }
      double avg = sum / cnt;
      StdOut.printf("Среднее — %.5f\n", avg);
   }
}
```

```
% java Average
1.23456
2.34567
3.45678
4.56789
<ctrl-d>
Среднее — 2.90123
```

Рис. 1.1.14. Пример работы клиентского класса из листинга 1.1.3

Перенаправление и конвейеры

Стандартные ввод и вывод позволяют воспользоваться расширениями командной строки, которые имеются во многих операционных системах. Добавив простую директиву к команде вызова программы, можно *перенаправить* ее стандартный вывод в файл — либо для долговременного хранения, либо для последующего ввода в другую программу (рис. 1.1.16):

```
% java RandomSeq 1000 100.0 200.0 > data.txt
```

public class **StdIn**			
static	boolean	isEmpty()	*true, если значений больше нет, иначе false*
static	int	readInt()	*чтение значения типа int*
static	double	readDouble()	*чтение значения типа double*
static	float	readFloat()	*чтение значения типа float*
static	long	readLong()	*чтение значения типа long*
static	boolean	readBoolean()	*чтение значения типа boolean*
static	char	readChar()	*чтение значения типа char*
static	byte	readByte()	*чтение значения типа byte*
static	String	readString()	*чтение значения типа String*
static	boolean	hasNextLine()	*есть ли во входном потоке еще одна строка?*
static	String	readLine()	*чтение остатка строки*
static	String	readAll()	*чтение остатка потока ввода*

Рис. 1.1.15. *API-интерфейс библиотеки статических методов для стандартного ввода*

Перенаправление из файла в стандартный ввод

% java Average < data.txt

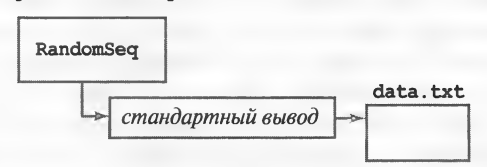
data.txt

стандартный ввод

Average

Перенаправление стандартного вывода в файл

% java RandomSeq 1000 100.0 200.0 > data.txt



Конвейер выходных данных одной программы на вход другой

% java RandomSeq 1000 100.0 200.0 | java Average

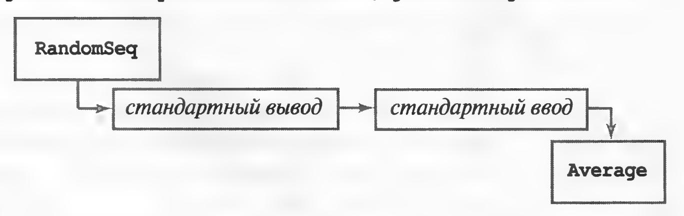


Рис. 1.1.16. *Перенаправление и конвейер, указываемые в командной строке*

Эта команда указывает, что поток стандартного вывода должен не выводиться в окно терминала, а записываться в текстовый файл по имени `data.txt`. Каждый вызов `StdOut.print()` или `StdOut.println()` добавляет текст в конец этого файла. В данном примере получается файл, содержащий 1000 случайных значений. В окно терминала ничего не выводится: значения направляются непосредственно в файл, имя которого указано после символа >. Так можно сохранить информацию для выборки в дальнейшем. При этом класс `RandomSeq` совершенно не меняется: он использует абстракцию стандартного вывода и никак не связан с изменением реализации этой абстракции. Аналогично можно перенаправить и стандартный ввод, чтобы методы библиотеки `StdIn` читали данные из файла, а не из приложения терминала:

```
% java Average < data.txt
```

Эта команда читает последовательность чисел из файла `data.txt` и вычисляет их среднее значение. Символ < представляет собой директиву, которая указывает операционной системе, что поток стандартного ввода следует реализовать с помощью чтения из текстового файла `data.txt`, а не ждать, пока пользователь введет что-то в окне терминала. Когда программа вызывает метод `StdIn.readDouble()`, операционная система читает очередное значение из файла. Совместное перенаправление вывода из одной программы на ввод другой программы называется ***конвейером***:

```
% java RandomSeq 1000 100.0 200.0 | java Average
```

Эта команда указывает, что стандартный вывод для `RandomSeq` и стандартный ввод для `Average` представляют собой один и тот же поток. В результате получается, как будто программа `RandomSeq` вводит генерируемые числа в окне терминала во время работы программы `Average`. Эта разница имеет серьезные последствия: она устраняет ограничение на размер возможных потоков ввода и вывода. Например, параметр `1000` в рассматриваемом примере можно заменить значением `1000000000`, хотя в компьютере может не быть памяти для хранения миллиарда чисел (правда, все так же необходимо время на их обработку).

Когда программа `RandomSeq` вызывает метод `StdOut.println()`, в конец потока добавляется строка, а когда программа `Average` вызывает метод `StdIn.readInt()`, из начала потока извлекается строка. Как это происходит во времени, зависит от операционной системы: она может выполнять некоторое время `RandomSeq`, пока не наберется какой-то объем выходных данных, а затем выполнять `Average`, чтобы потребить эти данные, либо может выполнять `Average`, пока ей не понадобятся входные данные, а затем переключиться на `RandomSeq`, чтобы она вывела необходимые выходные данные. Конечный результат будет одинаков, но нашим программам не нужно беспокоиться о таких деталях, т.к. они работают только с абстракциями стандартного ввода и стандартного вывода.

Ввод из файла и вывод в файл

В наших библиотеках `In` и `Out` имеются статические методы (рис. 1.1.17), которые реализуют абстракцию чтения из файла и записи в файл содержимого массива значений примитивного типа (или `String`). Это методы `readInts()`, `readDoubles()` и `readStrings()` из библиотеки `In` и методы `writeInts()`, `writeDoubles()` и `writeStrings()` из библиотеки `Out`. Аргумент `name` может быть файлом или веб-страницей. Например, это позволяет использовать файл и стандартный ввод для двух различных целей в одной программе, как сделано в программе `BinarySearch`.

Библиотеки `In` и `Out` также реализуют типы данных для методов экземпляров, которые позволяют более общим образом трактовать несколько файлов как потоки ввода и вывода, а веб-страницы — как потоки вывода, но об этом речь пойдет в разделе 1.2.

public class **In**			
static	int[]	readInts(String name)	*чтение значений* int
static	double[]	readDoubles(String name)	*чтение значений* double
static	String[]	readStrings(String name)	*чтение значений* String

public class **Out**			
static	void	write(int[] a, String name)	*запись значений* int
static	void	write(double[] a, String name)	*запись значений* double
static	void	write(String[] a, String name)	*запись значений* String

Примечание 1. Поддерживаются и другие примитивные типы.

Примечание 2. Поддерживаются библиотеки StdIn *и* StdOut *(нужно убрать аргумент* name*).*

Рис. 1.1.17. *API-интерфейсы статических методов для чтения и записи массивов*

Стандартный чертеж (основные методы)

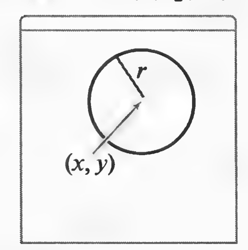
До этого момента наши абстракции ввода-вывода касались исключительно текстовых строк. А теперь мы познакомимся с абстракцией для генерации чертежей в качестве выходных данных. Эта легкая в применении библиотека позволяет воспользоваться визуальным носителем, чтобы получать гораздо больше информации, чем позволяет простой текст. Как и в случае стандартного ввода-вывода, наша абстракция стандартного чертежа реализована в библиотеке StdDraw; вы также можете загрузить файл StdDraw.java в свой рабочий каталог. Работать со стандартным чертежом очень просто: это как будто абстрактное чертежное устройство, которое может вычерчивать линии и точки на двумерном листе. Такое устройство может реагировать на команды формирования простых геометрических фигур, которые наши программы задают в виде вызовов статических методов из библиотеки StdDraw для вычерчивания линий, точек, текстовых строк, окружностей, прямоугольников и многоугольников. Как и методы для стандартного ввода и стандартного вывода, эти методы практически самодокументированны. Например, метод StdDraw.line() вычерчивает отрезок прямой линии, соединяющий точки (x_0, y_0) и (x_1, y_1), координаты которых заданы в аргументах. Метод StdDraw.point() вычерчивает точку (x, y), координаты которой заданы в аргументах, и т.д. (рис. 1.1.18). Геометрические фигуры можно залить (по умолчанию черным цветом). Стандартным полем чертежа является единичный квадрат (все координаты со значениями от 0 до 1). Стандартная реализация выводит поле чертежа на экране компьютера, с черными линиями и точками на белом фоне.

Стандартный чертеж (методы управления)

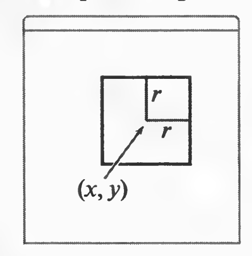
В библиотеке имеются также методы для изменения масштаба и размера листа, цвета и толщины линий, шрифта текста и длительности такта для анимированных чертежей (рис. 1.1.19 и 1.1.20). В качестве аргумента для метода setPenColor() можно указать один из цветов BLACK, BLUE, CYAN, DARK_GRAY, GRAY, GREEN, LIGHT_GRAY, MAGENTA, ORANGE, PINK, RED, BOOK_RED, WHITE и YELLOW, определенных как константы в библиотеке StdDraw (поэтому мы будем указывать их с помощью выражений вроде StdDraw.RED). Окно чертежа содержит также пункт меню, который позволяет сохранить чертеж в файле, в формате, удобном для публикации в сети.

```
StdDraw.point(x0, y0);
StdDraw.line(x1, y1, x2, y2);
```

```
StdDraw.circle(x, y, r);
```




```
StdDraw.square(x, y, r);
```




```
double[] x = {x0, x1, x2, x3};
double[] y = {y0, y1, y2, y3};
StdDraw.polygon(x, y);
```

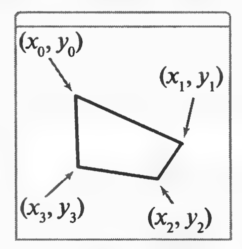


Рис. 1.1.18. Примеры применения методов библиотеки `StdDraw`

public class **StdDraw**	
static void	line(double x0, double y0, double x1, double y1)
static void	point(double x, double y)
static void	text(double x, double y, String s)
static void	circle(double x, double y, double r)
static void	filledCircle(double x, double y, double r)
static void	ellipse(double x, double y, double rw, double rh)
static void	filledEllipse(double x, double y, double rw, double rh)
static void	square(double x, double y, double r)
static void	filledSquare(double x, double y, double r)
static void	rectangle(double x, double y, double rw, double rh)
static void	filledRectangle(double x, double y, double rw, double rh)
static void	polygon(double[] x, double[] y)
static void	filledPolygon(double[] x, double[] y)

Рис. 1.1.19. API-интерфейс нашей библиотеки статических методов для стандартного чертежа (методы рисования)

public class **StdDraw**	
static void setXscale(double x0, double x1)	*установка диапазона x в (x0, x1)*
static void setYscale(double y0, double y1)	*установка диапазона y в (y0, y1)*
static void setPenRadius(double r)	*установка радиуса пера равным* r
static void setPenColor(Color c)	*установка цвета пера равным* c
static void setFont(Font f)	*установка шрифта текста в* f
static void setCanvasSize(int w, int h)	*задание ширины* w *и высоты* h *листа*
static void clear(Color c)	*очистка листа цветом* c
static void show(int dt)	*вывод всего с тактом* dt *миллисекунд*

Рис. 1.1.20. API-интерфейс нашей библиотеки статических методов для стандартного чертежа (методы управления)

В настоящей книге мы будем использовать библиотеку StdDraw для анализа данных и для создания визуальных представлений алгоритмов в действии. Несколько вариантов описаны в табл. 1.1.10; в тексте и упражнениях будут представлены еще многие другие примеры. Библиотека поддерживает и *анимацию* — понятно, что эта тема в основном освещается на сайте книги.

Таблица 1.1.10. Примеры использования библиотеки StdDraw

Данные	Реализация (фрагмент кода)	Результат
Значения функции	int N = 100; StdDraw.setXscale(0, N); StdDraw.setYscale(0, N*N); StdDraw.setPenRadius(.01); for (int i = 1; i <= N; i++) { StdDraw.point(i, i); StdDraw.point(i, i*i); StdDraw.point(i, i*Math.log(i)); }	
Массив случайных значений	int N = 50; double[] a = new double[N]; for (int i = 0; i < N; i++) a[i] = StdRandom.random(); for (int i = 0; i < N; i++) { double x = 1.0*i/N; double y = a[i]/2.0; double rw = 0.5/N; double rh = a[i]/2.0; StdDraw.filledRectangle(x, y, rw, rh); }	
Упорядоченный массив случайных значений	int N = 50; double[] a = new double[N]; for (int i = 0; i < N; i++) a[i] = StdRandom.random(); **Arrays.sort(a);** for (int i = 0; i < N; i++) { double x = 1.0*i/N; double y = a[i]/2.0; double rw = 0.5/N; double rh = a[i]/2.0; StdDraw.filledRectangle(x, y, rw, rh); }	

Бинарный поиск

Наша первая Java-программа, приведенная в листинге 1.1.4, основана на знаменитом, эффективном и широко распространенном алгоритме *бинарного поиска*. Этот пример является прототипом способа, которым мы будем рассматривать новые алгоритмы на протяжении книги. Как и все приводимые нами программы, он представляет собой точное определение метода и полную Java-реализацию, которую можно загрузить с сайта книги.

Листинг 1.1.4. Программа BinarySearch

```
import java.util.Arrays;
public class  BinarySearch
{
   public static int rank(int key, int[] a)
   {  // Массив должен быть отсортирован.
      int lo  = 0;
      int hi = a.length - 1;
      while (lo <= hi)
      {  // Key находится в a[lo..hi] или отсутствует.
         int mid = lo + (hi - lo) / 2;
         if      (key < a[mid]) hi = mid - 1;
         else if (key > a[mid]) lo = mid + 1;
         else                   return mid;
      }
      return -1;
   }
   public static void main(String[] args)
   {
      int[] whitelist = In.readInts(args[0]);
      Arrays.sort(whitelist);
      while (!StdIn.isEmpty())
      {  // Чтение значения key и вывод, если его нет в белом списке.
         int key = StdIn.readInt();
         if (rank(key, whitelist) < 0)
            StdOut.println(key);
      }
   }
}
```

Эта программа принимает в качестве аргумента имя файла с белым списком (последовательность целых чисел) и проверяет каждое значение, принятое из стандартного ввода, выводя в стандартный вывод только числа, отсутствующие в белом списке (рис. 1.1.21). Чтобы эффективно выполнять свою задачу, она использует алгоритм бинарного поиска, реализованный в статическом методе `rank()`. Полное обсуждение алгоритма бинарного поиска, доказательство его правильности, анализ производительности и области применения будут изложены в разделе 3.1.

```
% java BinarySearch tinyW.txt < tinyT.txt
50
99
13
```

Рис. 1.1.21. Пример работы программы `BinarySearch`

Алгоритм бинарного поиска

Подробному изучению алгоритма бинарного поиска посвящен раздел 3.2, а здесь мы приведем лишь краткое описание. Алгоритм реализован в статическом методе `rank()`, которому передаются в качестве аргументов целочисленный ключ `key` и *отсортированный* массив `a` целочисленных значений. Метод возвращает индекс ключа, если он присутствует в массиве, и -1 — если отсутствует. Он решает эту задачу с помощью переменных

tinyW.txt	tinyT.txt
84	23
48	50
68	10
10	99
18	18
98	23
12	98
23	84
54	11
57	10
48	48
33	77
16	13
77	54
11	98
29	77
	77
	68

отсутствуют в `tinyT.txt`

Рис. 1.1.22. *Небольшие тестовые файлы для клиента тестирования* `BinarySearch`

lo и hi, таких, что если ключ присутствует в массиве, то он находится в интервале a[lo..hi], и цикла, который проверяет средний элемент этого интервала (индекс mid). Если ключ равен a[mid], возвращается значение mid, иначе метод делит интервал пополам и просматривает левую половину, если ключ меньше a[mid], и правую — если больше. Этот процесс прекращается тогда, когда ключ найден или интервал пуст. Эффективность бинарного поиска объясняется тем, что ему необходимо проверить лишь несколько элементов массива (по сравнению с размером всего массива), чтобы обнаружить ключ или его отсутствие.

Клиент разработки

Для каждой реализации алгоритма мы включаем клиент разработчики main(), который можно использовать с демонстрационными входными файлами, представленными в книге и на сайте книги, чтобы лучше изучить алгоритм и оценить его производительность. В данном примере клиент читает целые числа из файла, имя которого задано в командной строке, затем выводит в стандартный вывод все числа, которые не присутствуют в файле. Мы используем небольшие тестовые файлы, наподобие приведенных на рис. 1.1.22, чтобы продемонстрировать их поведение и в качестве основы для трассировки и примеров, как на рис. 1.1.23. Мы используем большие тестовые файлы для моделирования реальных применений и тестирования производительности (рис. 1.1.24).



Рис. 1.1.23. *Бинарный поиск в упорядоченном массиве*

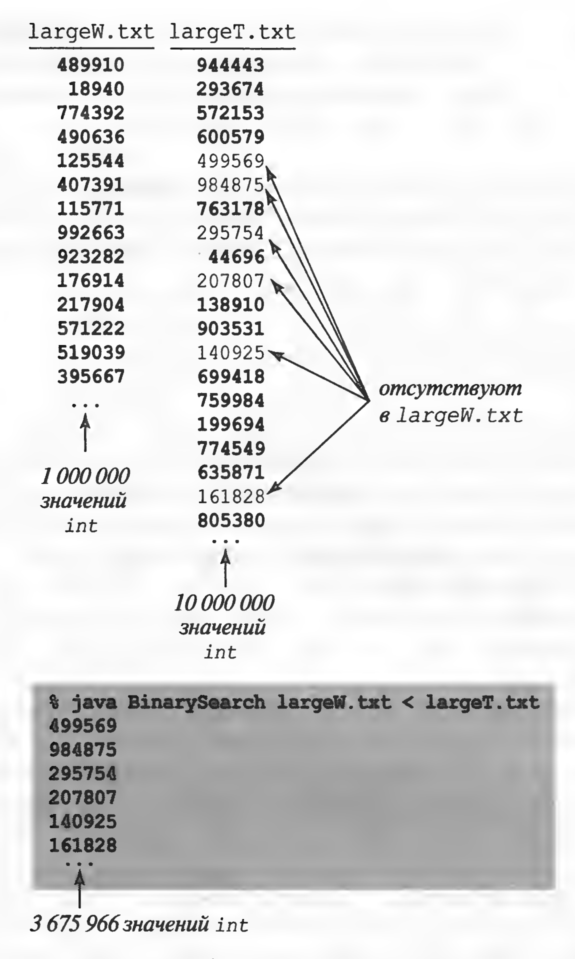


Рис. 1.1.24. Большие файлы для клиента тестирования `BinarySearch`

Белые списки

Если это возможно, наши клиенты разработки предназначены для имитации практических ситуаций и демонстрации необходимости конкретного алгоритма. В рассматриваемом случае процесс называется *фильтрацией белым списком* (whitelisting). Представьте себе банк, выдающий кредитные карточки, которому надо проверять, требует ли клиент выполнить транзакцию с верным счетом. Для этого можно предпринять следующие действия.

- Хранить номера счетов клиентов в файле, который мы будем называть *белым списком.*
- Выводить номер счета, связанный с каждой транзакцией, в поток стандартного ввода.
- Использовать клиент тестирования для вывода в стандартный вывод номеров, которые *не* связаны ни с каким клиентом. Предполагается, что банк будет отклонять такие транзакции.

В большом банке с миллионами клиентов вполне могут обрабатываться миллионы транзакций. Для моделирования такой ситуации на сайте книги приготовлены файлы `largeW.txt` (1 миллион целых чисел) и `largeT.txt` (10 миллионов чисел).

Производительность

Программа может быть работоспособной, но неэффективной. Например, можно написать гораздо более простую реализацию метода `rank()`, в которой осуществляется проверка каждого элемента (для этого даже не нужна упорядоченность массива):

```
public static int rank(int key, int[] a)
{
   for (int i = 0; i < a.length; i++)
      if (a[i] == key) return i;
   return -1;
}
```

Если имеется такое простое и понятное решение, зачем использовать сортировку слиянием и бинарный поиск? Если вы выполните упражнение 1.1.38, то увидите, что ваш компьютер слишком медленно работает, чтобы выполнять эту примитивную реализацию метода `rank()` для объемных входных данных (например, миллион элементов в белом списке и 10 миллионов транзакций). *Решение задачи белого списка для большого количества входных данных невозможно без эффективных алгоритмов, таких как бинарный поиск и сортировка слиянием.* Высокая производительность часто бывает жизненно необходима, поэтому в разделе 1.4 мы заложим основы для изучения производительности, и будем анализировать характеристики производительности всех наших алгоритмов — включая бинарный поиск (раздел 3.1) и сортировку слиянием (раздел 2.2).

В данном контексте цель при тщательном очерчивании нашей модели программирования состоит в том, чтобы иметь возможность выполнять код вроде `BinarySearch` на вашем компьютере, использовать его для тестирования данных, вроде предоставленных нами, и изменять его для моделирования разных ситуаций, наподобие описанных в упражнениях в конце данного раздела — и все это для того, чтобы лучше оценить применимость алгоритма. Кратко описанная нами модель программирования предназначена для выполнения таких действий, которые крайне важны для нашего способа изучения алгоритмов.

Перспектива

В этом разделе описана точная и полная модель программирования, которая служила на протяжении десятилетий (и все еще служит) многим программистам. Правда, современное программирование ушло на один шаг дальше. Этот следующий уровень называется *абстракцией данных*, а иногда *объектно-ориентированным программированием* — он будет описан в следующем разделе. Если говорить просто, то идея абстракции данных состоит в том, чтобы позволить программе определять *типы данных* (множества значений и множества операций над этими значениями), а не просто статические методы, которые оперируют с предопределенными типами данных.

Объектно-ориентированное программирование широко распространилось за последние десятилетия, а абстракция данных является основой для современной разработки программного обеспечения. Абстракция данных рассматривается в настоящей книге по трем основным причинам.

- Она расширяет возможность повторного использования кода с помощью модульного программирования. Например, наши алгоритмы сортировки в главе 2 и бинарный поиск и другие алгоритмы в главе 3 позволяют клиентам использовать один и тот же код для любых типов данных (не только целых чисел), в том числе и определенных клиентом.
- Она предоставляет удобный механизм для построения так называемых *связных* структур данных, которые более гибки по сравнению с массивами и представляют собой основу для эффективных алгоритмов во многих ситуациях.
- Она позволяет точно определить алгоритмические сложности, с которыми мы сталкиваемся. Например, наш алгоритм объединения-поиска, рассмотренный в разделе 1.5, алгоритмы очередей с приоритетами из раздела 2.4 и алгоритмы работы с таблицами имен из главы 3 — все они ориентированы на определение структур данных, которые делают возможными эффективные реализации *множества* операций. Эта задача в точности эквивалентна абстракции данных.

Но, несмотря на все эти соображения, основное внимание мы будем уделять изучению алгоритмов. В этом контексте мы перейдем к рассмотрению важных особенностей объектно-ориентированного программирования, которые относятся к нашей миссии.

Вопросы и ответы

Вопрос. **Что такое байт-код Java?**

Ответ. Низкоуровневая версия программы, которая выполняется на *виртуальной машине* Java. Этот уровень абстракции облегчает разработчикам распространение Java-программ на широком диапазоне устройств.

Вопрос. **Пожалуй, плохо, что в Java допустимы переполнения при обработке значений `int`, которые дают неверные значения. Нельзя ли в Java встроить автоматическую проверку на переполнение?**

Ответ. По этому вопросу у программистов нет единого мнения. Кратко можно сказать, что отсутствие такой проверки является одной из причин того, что такие типы называются *примитивными*. Чтобы избежать подобных проблем, необходим немного больший объем знаний. Мы используем тип `int` для не очень больших чисел (менее десяти десятичных цифр) и тип `long`, если значения могут доходить до миллиардов и более.

Вопрос. **Каково значение `Math.abs(-2147483648)`?**

Ответ. `-2147483648`. Этот странный (хотя и верный) результат — типичный пример эффекта целочисленного переполнения.

Вопрос. **Как можно инициализировать переменную `double` значением бесконечности?**

Ответ. В Java для этого имеются встроенные константы: `Double.POSITIVE_INFINITY` и `Double.NEGATIVE_INFINITY`.

Вопрос. **Можно ли сравнивать значения типов `double` и `int`?**

Ответ. Только после преобразования типа, но Java обычно выполняет необходимое преобразование автоматически. Например, если `x` — переменная типа `int` со значением `3`, то выражение `(x < 3.1)` равно `true`: перед сравнением Java преобразует `x` в тип `double` (поскольку `3.1` имеет тип `double`).

Вопрос. **Что произойдет, если использовать переменную до ее инициализации значением?**

Ответ. Java сообщит об ошибке на этапе компиляции, если в коде существует хотя бы какой-то путь, который может привести к использованию неинициализированной переменной.

Вопрос. **Чему равны значения `1/0` и `1.0/0.0` в выражениях Java?**

Ответ. Первое из них генерирует *исключение* времени выполнения по причине деления на ноль (что приводит к завершению программы, т.к. соответствующее значение не определено), а второе имеет значение `Infinity`.

Вопрос. **Можно ли использовать операции < и > для сравнения переменных `String`?**

Ответ. Нет. Эти операции определены только для примитивных типов.

Вопрос. **Чему равен результат деления и остаток для отрицательных целых чисел?**

Ответ. Частное `a/b` округляется в сторону `0`; остаток `a % b` определен так, что `(a / b) * b + a % b` всегда равно `a`. Например, выражения `-14/3` и `14/-3` оба равны `-4`, но `-14 % 3` равно `-2`, а `14 % -3` равно `2`.

Вопрос. **Почему нужно писать `(a && b)`, а не просто `(a & b)`?**

Ответ. Операции `&`, `|` и `^` — это *поразрядные* логические операции для целых типов, которые выполняют операции *и*, *или* и *исключающее или* (соответственно) в позиции каждого разряда. Поэтому выражение `10|6` равно `14`, а выражение `10^6` равно `12`. Эти операции также (хотя и редко) используются в данной книге. Операции `&&` и `||` применимы только к логическим выражениям и существуют отдельно из-за свойства *сокращенного вычисления*: выражение вычисляется слева направо лишь до тех пор, пока не станет известно его значение.

Вопрос. **Является ли проблемой неоднозначность вложенных операторов `if`?**

Ответ. Да. В Java запись

```
if <выражение1> if <выражение2> <операторA> else <операторB>
```

эквивалентна

```
if <выражение1> { if <выражение2> <операторA> else <операторB> }
```

даже если подразумевалось

```
if <выражение1> { if <выражение2> <операторA> } else <операторB>
```

Применение дополнительных фигурных скобок позволяет избежать этого эффекта *висячего* `else`.

Вопрос. **Чем цикл `for` отличается от его формулировки с помощью цикла `while`?**

Ответ. Код в заголовке цикла `for` считается принадлежащим блоку тела цикла. В типичном цикле `for` инкрементируемая переменная недоступна для использования в последующих операторах, но в аналогичном цикле `while` она доступна. Это различие часто является причиной использования цикла `while` вместо `for`.

Вопрос. **Некоторые программисты на Java при объявлении массивов пишут `int a[]` вместо `int[] a`. В чем разница?**

Ответ. В Java оба эти способа допустимы и эквивалентны. Первый способ взят из C. Второй используется в Java чаще, т.к. тип переменной `int[]` нагляднее показывает, что это *массив* целых чисел.

Вопрос. **Почему элементы массивов нумеруются с 0, а не с 1?**

Ответ. Это соглашение берет начало от программирования на машинном языке, где адрес элемента массива вычисляется сложением индекса и адреса начала массива. Нумерация с 1 приводит либо к появлению пустого места в начале массива, либо к затратам времени на вычитание 1.

Вопрос. **Если `a[]` — массив, то почему вызов `StdOut.println(a)` выводит шестнадцатеричное число вроде `@f62373`, а не элементы массива?**

Ответ. Хороший вопрос. Такой вызов выводит адрес массива в памяти, хотя вряд ли это то, что требовалось.

Вопрос. **Почему мы не используем стандартные Java-библиотеки для ввода и графики?**

Ответ. Мы *используем* их, но предпочитаем работать с более простыми абстрактными моделями. Java-библиотеки `StdIn` и `StdDraw` созданы для производственного программирования, а стандартные библиотеки и их API-интерфейсы несколько громоздки. Чтобы ознакомиться с ними, просмотрите код в файлах `StdIn.java` и `StdDraw.java`.

Вопрос. **Может ли программа повторно ввести данные из стандартного ввода?**

Ответ. Нет. У вас только одна попытка — так же, как невозможно отменить `println()`.

Вопрос. **Что произойдет, если программа попытается прочитать данные, когда стандартный ввод уже пуст?**

Ответ. Будет получена ошибка. Ее можно избежать ее с помощью метода `StdIn.isEmpty()`, который проверяет, есть ли еще данные для ввода.

Вопрос. **Что значит такое сообщение об ошибке:**

```
Exception in thread "main" java.lang.NoClassDefFoundError: StdIn
```

Ответ. Вы, видимо, забыли поместить файл `StdIn.java` в свой рабочий каталог.

Вопрос. **Может ли статический метод в Java принять в качестве аргумента другой статический метод?**

Ответ. Нет. Хороший вопрос, т.к. многие другие языки допускают такие аргументы.

Упражнения

1.1.1. Чему равно значение каждого из следующих выражений?

а) `( 0 + 15 ) / 2`

б) `2.0e-6 * 100000000.1`

в) `true && false || true && true`

1.1.2. Чему равно значение каждого из следующих выражений?

а) `(1 + 2.236)/2`

б) `1 + 2 + 3 + 4.0`

в) `4.1 >= 4`

г) `1 + 2 + "3"`

1.1.3. Напишите программу, которая принимает из командной строки три целочисленных аргумента и выводит строку `равны`, если все они равны, и `не равны` в противном случае.

3 Зak. 3239

1.1.4. Что неправильно (если неправильно) в каждом из следующих операторов?

а) `if (a > b) then c = 0;`

б) `a > b { c = 0; }`

в) `(a > b) c = 0;`

г) `(a > b) c = 0 else b = 0;`

1.1.5. Напишите фрагмент кода, который выводит `true`, если значения обеих переменных `x` и `y` типа `double` находятся строго между 0 и 1, и `false` в остальных случаях.

1.1.6. Что выведет следующая программа?

```
int f = 0;
int g = 1;
for (int i = 0; i <= 15; i++)
{
   StdOut.println(f);
   f = f + g;
   g = f - g;
}
```

1.1.7. Какое значение выведет каждый из следующих фрагментов кода?

а)
```
double t = 9.0;
while (Math.abs(t - 9.0/t) > .001)
   t = (9.0/t + t) / 2.0;
StdOut.printf("%.5f\n", t);
```

б)
```
int sum = 0;
for (int i = 1; i < 1000; i++)
   for (int j = 0; j < i; j++)
      sum++;
StdOut.println(sum);
```

в)
```
int sum = 0;
for (int i = 1; i < 1000; i *= 2)
   for (int j = 0; j < N; j++)
      sum++;
StdOut.println(sum);
```

1.1.8. Что выведет каждый из следующих операторов?

а) `System.out.println('b');`

б) `System.out.println('b' + 'c');`

в) `System.out.println((char) ('a' + 4));`

1.1.9. Напишите фрагмент кода, который помещает двоичное представление положительного целого `N` в переменную `String s`.

Решение. В Java для этого имеется встроенный метод `Integer.toBinaryString(N)`, но цель упражнения — понять, как можно реализовать такой метод. Вот очень лаконичное решение:

```
String s = "";
for (int n = N; n > 0; n /= 2)
   s = (n % 2) + s;
```

1.1.10. Что неверно в следующем фрагменте кода?

```
int[] a;
for (int i = 0; i < 10; i++)
   a[i] = i * i;
```

Решение. Здесь не выделяется память для a[] с помощью операции new. Этот код приведет к появлению ошибки variable a might not have been initialized (переменная a может быть не инициализирована).

1.1.11. Напишите фрагмент кода, который выводит содержимое двумерного логического массива, используя звездочки для представления значений true и пробелы — для значений false. Добавьте номера строк и столбцов.

1.1.12. Что выведет следующий фрагмент кода?

```
int[] a = new int[10];
for (int i = 0; i < 10; i++)
   a[i] = 9 - i;
for (int i = 0; i < 10; i++)
   a[i] = a[a[i]];
for (int i = 0; i < 10; i++)
   System.out.println(i);
```

1.1.13. Напишите фрагмент кода для вывода *транспозиции* двумерного массива с *M* строками и *N* столбцами (строки заменены столбцами).

1.1.14. Напишите статический метод lg(), который принимает в качестве аргумента значение N типа int и возвращает наибольшее целое число, не большее, чем двоичный логарифм N. *Не* используйте библиотеку Math.

1.1.15. Напишите статический метод histogram(), который принимает в качестве аргументов массив a[] значений типа int и целое число M и возвращает массив длиной M, i-й элемент которого равен количеству появлений числа i в массиве-аргументе. Если значения в a[] находятся между 0 и M-1, то сумма значений полученного массива должна быть равна a.length.

1.1.16. Приведите значение exR1(6):

```
public static String exR1(int n)
{
   if (n <= 0) return "";
   return exR1(n-3) + n + exR1(n-2) + n;
}
```

1.1.17. Чем плоха следующая рекурсия?

```
public static String exR2(int n)
{
   String s = exR2(n-3) + n + exR2(n-2) + n;
   if (n <= 0) return "";
   return s;
}
```

Ответ. База рекурсии никогда не будет достигнута. Вызов exR2(3) приведет к вызовам exR2(0), exR2(-3), exR3(-6) и т.д., пока не возникнет ошибка StackOverflowError.

1.1.18. Имеется следующая рекурсивная функция:

```
public static int mystery(int a, int b)
{
```

```
  if (b == 0)     return 0;
  if (b % 2 == 0) return mystery(a+a, b/2);
  return mystery(a+a, b/2) + a;
}
```

Чему равны значения `mystery(2, 25)` и `mystery(3, 11)`? Какое значение вычисляет вызов `mystery(a, b)` для положительных целых аргументов `a` и `b`? Замените операцию + операцией * и оператор `return 0` оператором `return 1` и снова ответьте на этот вопрос.

1.1.19. Выполните на своем компьютере следующую программу:

```
public class Fibonacci
{
  public static long F(int N)
  {
    if (N == 0) return 0;
    if (N == 1) return 1;
    return F(N-1) + F(N-2);
  }
  public static void main(String[] args)
  {
    for (int N = 0; N < 100; N++)
      StdOut.println(N + " " + F(N));
  }
}
```

Каково наибольшее значение `N`, для которого программа вычисляет значение `F(N)` менее чем за час? Разработайте лучшую реализацию `F(N)`, которая сохраняет вычисленные значения в массиве.

1.1.20. Напишите рекурсивный статический метод, который вычисляет значение $\ln(N!)$.

1.1.21. Напишите программу, которая читает строки из стандартного ввода, где каждая строка содержит имя и два целых числа, а затем использует метод `printf()` для вывода таблицы, в каждой строке которой содержатся имя, оба целых числа и результат деления первого числа на второе с точностью до трех десятичных знаков. Похожую программу можно использовать для вывода рейтингов игроков в бейсбол или оценок студентов.

1.1.22. Напишите версию программы `BinarySearch`, в которой задействован рекурсивный метод `rank()`, приведенный в листинге 1.1.1, и выполняется *трассировка* вызовов этого метода. При каждом вызове рекурсивного метода он должен выводить значения `lo` и `hi` с отступом слева, соответствующим глубине рекурсии. *Совет*: добавьте в метод дополнительный аргумент, который будет отслеживать глубину рекурсии.

1.1.23. Добавьте в клиент тестирования `BinarySearch` возможность реагировать на второй аргумент: если он положителен, то должен выводить числа из стандартного ввода, которые *отсутствуют* в белом списке, а если отрицателен — которые *присутствуют* там.

1.1.24. Приведите последовательность значений p и q, вычисляемые при работе алгоритма Евклида, когда он находит наибольший общий делитель 105 и 24. Напишите на основе кода, приведенного на рис. 1.0.1, программу `Euclid`, которая принимает из командной строки два целых числа и вычисляет их наи-

больший общий делитель, выводя оба аргумента для каждого вывода рекурсивного метода. Используйте полученную программу для вычисления наибольшего общего делителя 1111111 и 1234567.

1.1.25. Докажите с помощью математической индукции, что алгоритм Евклида вычисляет наибольший общий делитель для любой пары неотрицательных целых чисел p и q.

Творческие задачи

1.1.26. *Сортировка трех чисел.* Допустим, переменные a, b, c и t имеют один и тот же числовой примитивный тип. Покажите, что следующий код располагает значения a, b и c в порядке возрастания:

```
if (a > b) { t = a; a = b; b = t; }
if (a > c) { t = a; a = c; c = t; }
if (b > c) { t = b; b = c; c = t; }
```

1.1.27. *Биномиальное распределение.* Оцените количество рекурсивных вызовов, которые потребуются коду

```
public static double binomial(int N, int k, double p)
{
   if (N == 0 && k == 0) return 1.0;
   if (N < 0 || k < 0) return 0.0;
   return (1.0 - p)*binomial(N-1, k, p) + p*binomial(N-1, k-1, p);
}
```

для вычисления значения binomial(100, 50). Разработайте лучшую реализацию, в которой вычисленные значения хранятся в массиве.

1.1.28. *Удаление дубликатов.* Измените клиент тестирования в программе BinarySearch, чтобы после сортировки белого списка из него удалялись все повторяющиеся ключи.

1.1.29. *Равные ключи.* Добавьте в программу BinarySearch статический метод rank(), который принимает в качестве аргументов ключ и упорядоченный массив целочисленных значений (некоторые могут повторяться) и возвращает количество элементов, меньших ключа. Добавьте также аналогичный метод count(), который подсчитывает количество элементов, равных нулю. ***Примечание***: если i и j — значения, возвращенные вызовами rank(key, a) и count(key, a) соответственно, то a[i..i+j-1] — значения в массиве, равные key.

1.1.30. *Работа с массивом.* Напишите фрагмент кода, который создает логический массив a[][] размером $N \times N$ — такой, что a[i][j] равно true, если i и j взаимно просты (не имеют общего делителя), и false в противном случае.

1.1.31. *Случайные соединения.* Напишите программу, которая принимает из командной строки в качестве аргументов целое N и значение p типа double (между 0 до 1), вычерчивает N точек размера 0.05, равномерно распределенных по окружности, и затем вычерчивает серые отрезки, соединяющие каждую пару точек с вероятностью p.

1.1.32. *Гистограмма.* Пусть поток стандартного ввода содержит последовательность чисел типа double. Напишите программу, которая принимает из командной строки целое число N и два вещественных значения l и r и использует библиотеку

StdDraw для вывода гистограммы количеств чисел, которые попали в каждый из интервалов, полученных делением интервала (l, r) на N меньших интервалов равного размера.

1.1.33. *Библиотека матричных вычислений.* Напишите библиотеку Matrix, которая реализует следующий API-интерфейс:

public class **Matrix**

static	double	dot(double[] x, double[] y)	*скалярное произведение векторов*
static	double[][]	mult(double[][] a, double[][] b)	*произведение двух матриц*
static	double[][]	transpose(double[][] a)	*транспонирование*
static	double[]	mult(double[][] a, double[] x)	*произведение матрицы на вектор*
static	double[]	mult(double[] y, double[][] a)	*произведение вектора на матрицу*

Разработайте клиент тестирования, который читает значения из стандартного ввода и проверяет работу всех методов.

1.1.34. *Фильтрация.* Какие действия из приведенных ниже *требуют* сохранения всех значений из стандартного ввода (например, в массиве), а которые можно реализовать в виде фильтра, используя лишь фиксированное количество переменных и массив фиксированного размера (не зависящего от N)? Для каждого из действий входные данные поступают из стандартного ввода и содержат N вещественных чисел между 0 и 1.

- Вывод максимального и минимального значений.
- Вывод медианы чисел.
- Вывод k-го наименьшего значения, для $k < 100$.
- Вывод суммы квадратов всех чисел.
- Вывод среднего значения N чисел.
- Вывод процента чисел, которые больше среднего значения.
- Вывод N чисел в порядке возрастания.
- Вывод N чисел в случайном порядке.

Эксперименты

1.1.35. *Моделирование игральных костей.* Следующий код вычисляет точное распределение вероятности для суммы очков при двух бросаниях костей:

```
int SIDES = 6;
double[] dist = new double[2*SIDES+1];
for (int i = 1; i <= SIDES; i++)
   for (int j = 1; j <= SIDES; j++)
      dist[i+j] += 1.0;
for (int k = 2; k <= 2*SIDES; k++)
   dist[k] /= 36.0;
```

Значение dist[i] — это вероятность того, что сумма очков двух костей равна k. Проведите эксперименты для проверки моделирования N бросков костей, подсчитывая частоты выпадений каждого значения при вычислении суммы двух случайных целых чисел от 1 до 6. Насколько большим должно быть N, чтобы эмпирические результаты соответствовали теоретическим с точностью до трех десятичных знаков?

1.1.36. *Эмпирическая проверка перемешиваний.* Выполните вычислительные эксперименты для проверки, что код перемешивания в табл. 1.1.6 работает так, как заявлено. Напишите программу ShuffleTest, которая принимает из командной строки аргументы M и N, выполняет N перемешиваний массива размером M, который инициализируется значениями a[i] = i перед каждым перемешиванием, и выводит таблицу $M \times M$, в которой строка i содержит количество раз, когда i встретилось в позиции j для всех j. Все элементы массива должны быть приблизительно равны N/M.

1.1.37. *Неправильное перемешивание.* Допустим, что в нашем коде перемешивания выбирается случайное число не между i и N-1, а между 0 и N-1. Покажите, что полученный порядок *не* в точности равен одной из $N!$ возможностей. Выполните тесты из предыдущего упражнения для этой версии.

1.1.38. *Сравнение бинарного поиска и примитивного поиска.* Напишите программу BruteForceSearch, которая использует примитивный метод поиска, описанный в разделе “Производительность”, и сравните время его выполнения на вашем компьютере со временем работы BinarySearch для файлов largeW.txt и largeT.txt.

1.1.39. *Случайные сопоставления.* Напишите клиент BinarySearch, который принимает в качестве аргумента командной строки значение T типа int и выполняет T повторений следующего эксперимента для $N = 10^3$, 10^4, 10^5 и 10^6: генерируются два массива N случайно сгенерированных положительных шестизначных целых чисел и находится количество чисел, которые присутствуют в обоих массивах. Выведите таблицу со средним количеством этой величины для T повторений и для каждого значения N.

1.2. Абстракция данных

Тип данных — это множество значений и множество операций над этими значениями. В предыдущем разделе мы подробно рассмотрели *примитивные* типы данных — например, *значения* примитивного типа данных `int` представляют собой целые числа от -2^{31} до $2^{31}-1$, а к *операциям* над ними относятся `+`, `*`, `-`, `/`, `%`, `<` и `>`. В принципе, все наши программы можно было бы написать с помощью только встроенных примитивных типов, но гораздо удобнее делать это на более высоком уровне абстракции. В данном разделе мы рассмотрим процесс определения и использования типов данных, который называются *абстракцией данных* (и дополняет стиль *абстракции функций*, который был описан в разделе 1.1).

Программирование на Java в значительной мере основано на построении типов данных, называемых *ссылочными типами*, с помощью широко известных классов Java. Этот стиль программирования носит название *объектно-ориентированного программирования*, поскольку в его основе лежит концепция *объекта* — сущности, которая содержит значение типа данных. Работая с примитивными типами Java, мы, по сути, ограничены программами, которые обрабатывают числа, а ссылочные типы позволяют писать программы, работающие со строками, изображениями, звуками и сотнями других абстракций, которые доступны в стандартных Java-библиотеках и на сайте книги. Но еще важнее, чем библиотеки или предопределенные типы данных, то, что *мы можем определять собственные типы данных* для реализации любой необходимой абстракции.

Абстрактный тип данных (АТД) — это тип данных, представление которого скрыто от клиента. Реализация АТД в виде Java-класса не очень отличается от реализации библиотеки функций в виде набора статических методов. Основное различие в том, что мы связываем *данные* с реализацией функций и скрываем представление данных от клиента. При *использовании* АТД основное внимание уделяется *операциям*, указанным в API-интерфейсе, при этом представление данных не рассматривается; а при *реализации* АТД основное внимание уделяется *данным*, а затем реализуются операции над этими данными.

Абстрактные типы данных важны потому, что они поддерживают инкапсуляцию при проектировании программы. В настоящей книге они используются как средство для:

- точного описания задач в форме API-интерфейсов для использования различными клиентами;
- описания алгоритмов и структур данных в виде реализаций API-интерфейсов.

Основной причиной изучения различных алгоритмов решения одной и той же задачи является различие характеристик производительности этих алгоритмов. Абстрактные типы данных представляют собой удобную среду для изучения алгоритмов, т.к. они позволяют немедленно использовать знания о производительности этих алгоритмов: один алгоритм можно просто заменить другим с целью увеличения производительности для всех клиентов без необходимости изменения клиентского кода.

Использование абстрактных типов данных

Для использования типа данных не обязательно знать, как он реализован — поэтому мы сейчас расскажем, как писать программы, использующие простой тип данных `Counter`, значения которых представляют собой название и неотрицательное целое число.

Операции, доступные для этого типа данных — *создание с обнулением*, *увеличение на единицу* и *проверка текущего значения*. Эта абстракция полезна во многих контекстах. Например, ее удобно было бы использовать в программном обеспечении электронного голосования, чтобы голосующий мог просто увеличить счетчик указанного им кандидата на единицу, или же для подсчета фундаментальных операций при анализе производительности алгоритмов. Для использования такой абстракции необходимо ознакомиться с механизмом для указания операций с типом данных и с механизмом языка Java для создания и обработки значений типа данных. Такие механизмы крайне важны в современном программировании, и они будут применяться во всей книге — поэтому наш первый пример следует рассмотреть со всем вниманием.

API-интерфейс для абстрактного типа данных

Для описания поведения абстрактного типа данных мы будем использовать *интерфейс прикладного программирования* (application programming interface — API), который представляет собой список *конструкторов* и *методов экземпляров* (операций) с неформальным описанием действия каждого из них. Пример API-интерфейса для типа данных `Counter` приведен на рис. 1.2.1.

public class **Counter**		
	`Counter(String id)`	*создание счетчика с именем* `id`
`void`	`increment()`	*увеличение счетчика на единицу*
`int`	`tally()`	*количество увеличений с момента создания*
`String`	**`toString()`**	*строковое представление*

Рис. 1.2.1. *API-интерфейс счетчика*

Хотя основу определения типа данных составляет множество значений, роль этих значений не видна из API-интерфейса — видны только действия над ними. Поэтому определение АТД имеет общие черты с библиотекой статических методов (см. подраздел “Свойства методов” в предыдущем разделе).

- Оба реализуются в виде Java-класса.
- Методы экземпляров могут принимать ноль или более аргументов указанного типа, разделенных запятыми и заключенных в скобки.
- Они могут возвращать значение указанного типа или не возвращать значения (обозначается ключевым словом `void`).

А вот три существенных различия.

- Имя некоторых элементов API-интерфейса совпадает с именем класса, и у них не задан возвращаемый тип. Такие элементы называются *конструкторами* и имеют специальное предназначение. Конструктор класса `Counter` принимает аргумент типа `String`.
- У методов экземпляров нет модификатора `static`. Это *не* статические методы, они предназначены для действий со значениями типов данных.
- Некоторые методы экземпляров присутствуют, чтобы удовлетворять соглашениям Java — такие методы мы называем *унаследованными* и выделяем в API-интерфейсе серым шрифтом.

Как и API-интерфейсы для библиотек статических методов, API-интерфейс для абстрактного типа данных представляет собой контракт со всеми клиентами и, таким образом, отправную точку как для разработки кода любого клиента, так и для разработки реализации типа данных. В нашем случае API-интерфейс указывает, что для использования типа `Counter` имеются конструктор `Counter()`, методы экземпляров `increment()` и `tally()` и унаследованный метод `toString()`.

Унаследованные методы

Различные соглашения, принятые в Java, позволяют типу данных воспользоваться встроенными языковыми механизмами, помещая конкретные методы в API-интерфейс. Например, все типы данных Java *наследуют* метод `toString()`, который возвращает символьное представление значений типа данных. Java вызывает этот метод, когда значение любого типа данных необходимо конкатенировать со значением типа `String` с помощью операции +. Стандартная реализация не особенно полезна (она выдает строковое представление адреса памяти, где находится значение типа данных), поэтому мы часто будем предоставлять реализацию, переопределяющую стандартную, и при этом добавлять метод `toString()` в API-интерфейс. Другие примеры таких методов — `equals()`, `compareTo()` и `hashCode()` (см. табл. 1.2.4).

Код клиента

Как и модульное программирование, основанное на статических методах, API-интерфейс позволяет писать клиентский код без знания деталей реализации (и писать код реализации, не зная деталей любого конкретного клиента). Механизмы организации программ в виде независимых модулей, описанные в разделе 1.1, полезны для всех Java-классов и поэтому эффективны для модульного программирования с АТД так же, как и для библиотек статических методов. Поэтому АТД можно использовать в любой программе — при условии, что его исходный код находится в `.java`-файле в том же каталоге, в стандартной Java-библиотеке, доступен с помощью оператора `import` либо с помощью одного из механизмов пути класса, описанных на сайте книги. Инкапсуляция всего кода, реализующего тип данных, в одном Java-классе позволяет разрабатывать код клиента на более высоком уровне абстракции. Для разработки клиентского кода необходима возможность *объявлять переменные*, *создавать объекты* для хранения значений типа данных и *обеспечить доступ* к значениям методам экземпляров, чтобы оперировать этими значениями. Такие процессы отличаются от соответствующих процессов для примитивных типов, хотя имеется и много похожих моментов.

Объекты

То, что переменная `heads` должна быть связана с данными типа `Counter`, можно объявить с помощью естественного кода

```
Counter heads;
```

Но как присваивать значения и указывать операции? Ответ на этот вопрос опирается на фундаментальную концепцию в абстракции данных: *объект* — это сущность, которая может принимать значение типа данных. Объекты характеризуются тремя неотъемлемыми свойствами: *состояние*, *идентичность* и *поведение*. *Состояние* объекта — это значение из его типа данных. *Идентичность* объекта отличает один объект от другого. Идентичность объекта удобно рассматривать как место в памяти, где хранится его значение. *Поведение* объекта — это эффект операций типа данных. Реализация отвечает

только за поддержку идентичности объекта, чтобы клиентский код мог использовать тип данных независимо от представления его состояния, учитывая только API-интерфейс с описанием поведения объекта. Состояние объекта можно использовать для предоставления информации клиенту, для вызова побочного эффекта или для изменения с помощью одной из операций типа данных, но детали представления значения типа данных не важны клиентскому коду. *Ссылка* — механизм для доступа к объекту. Номенклатура языка Java содержит четкое отличие от примитивных типов (где переменные связаны со значениями): не примитивные типы называются *ссылочными типами*. Детали реализации ссылок различаются в разных реализациях Java, но ссылки удобно представлять как адрес памяти, как показано на рис. 1.2.2 (для краткости на нем используются трехзначные адреса памяти).

Один объект Counter





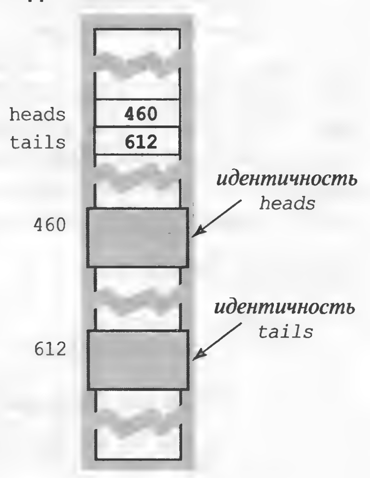


Рис. 1.2.2. Представление объектов

Создание объектов

Каждое значение типа данных хранится в некотором объекте. Для создания отдельного объекта (*экземпляра*) вызывается конструктор. Для этого записывается ключевое слово `new`, за ним имя класса, и потом `()` (или список аргументов в скобках, если конструктор принимает аргументы), как показано на рис. 1.2.3. Конструктор не имеет возвращаемого типа, т.к. он всегда возвращает ссылку на объект его типа данных.

При каждом вызове `new()` система:

- выделяет в памяти место для объекта;
- вызывает конструктор для инициализации его значения;
- возвращает ссылку на объект.

В клиентском коде объекты обычно создаются с помощью инициализирующего объявления, которое связывает переменную с объектом — как это зачастую делается с переменными примитивных типов. Но, в отличие от примитивных типов, переменные связываются со ссылками на объекты, а не с самими значениями типа данных.

Можно создать любое количество объектов одного и того же класса: каждый объект обладает собственной идентичностью и может содержать такое же значение, как и другой объект того же типа (или другое).

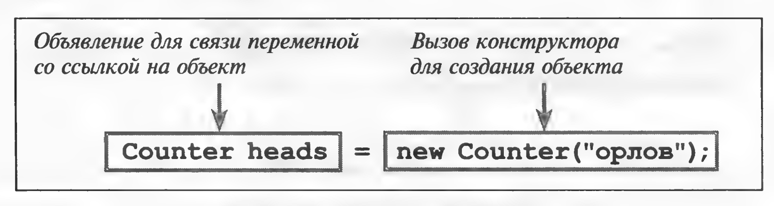


Рис. 1.2.3. Создание объекта

Например, код

```
Counter heads = new Counter("орлов");
Counter tails = new Counter("решек");
```

создает два различных объекта `Counter`. В абстрактном типе данных детали представления значения скрыты от клиентского кода. Можно предположить, что значение, связанное с каждым объектом `Counter`, состоит из названия типа `String` и счетчика типа `int`, но *нельзя писать код, который зависит от любого конкретного представления* (или даже знает, что это предположение верно — возможно, счетчик имеет тип `long`).

Вызов методов экземпляров

Назначение метода экземпляра — в оперировании со значениями типа данных, поэтому в Java имеется специальный механизм для вызова методов экземпляров, который подчеркивает связь с объектом. А именно: в вызове метода экземпляра записывается имя переменной, которая указывает на объект, потом точка, потом имя метода экземпляра, а за ним ноль или более аргументов, заключенных в скобки и разделяемых запятыми (рис. 1.2.4). Метод экземпляра может *изменить* значение типа данных или просто *узнать* его значение. Методы экземпляров имеют все свойства статических методов, которые были рассмотрены в разделе 1.1 (подраздел "Свойства методов") — аргументы передаются по значению, имена методов могут быть перегружены, они могут возвращать значение и вызывать побочные эффекты — но у них есть характерное дополнительное свойство: *каждый вызов связан с некоторым объектом*. Например, код

```
heads.increment();
```

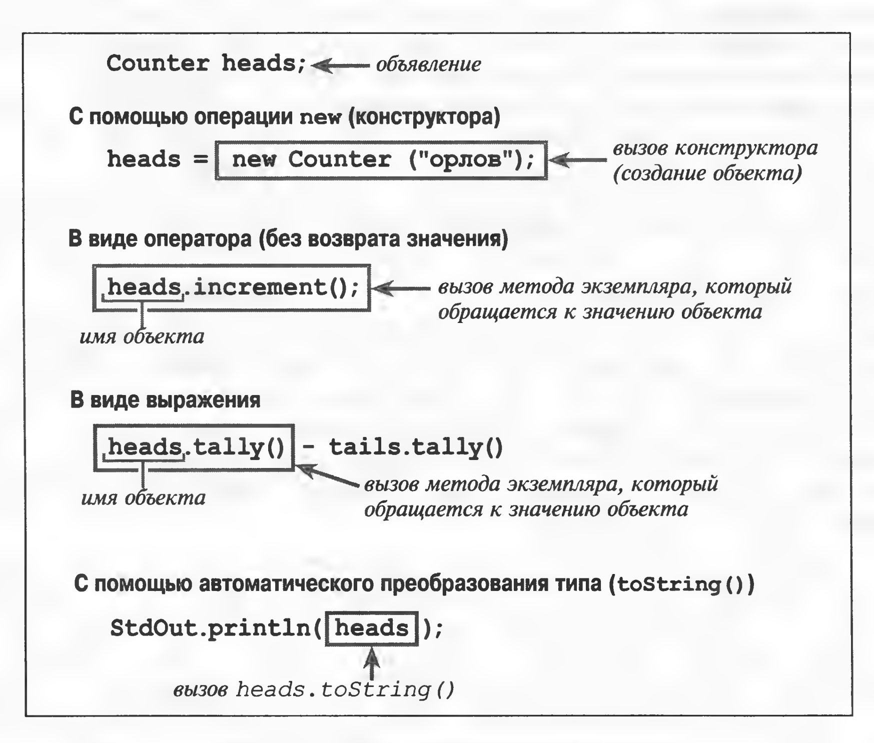


Рис. 1.2.4. Вызов методов экземпляров

вызывает метод экземпляра `increment()` для работы с объектом `heads` типа `Counter` (в данном случае выполняется увеличение счетчика), а код

```
heads.tally() - tails.tally();
```

вызывает метод `tally()` дважды: первый раз для работы с объектом `heads` типа `Counter`, а затем для работы с объектом `tails` типа `Counter` (в данном случае возвращается целочисленное значение счетчика). Как видно из этих примеров, вызовы методов экземпляров в коде клиента ничем не отличаются от вызовов статических методов — как в виде операторов (`void`-методы), так и в виде значений в выражениях (методы, возвращающие значения). Основное назначение статических методов — реализация функций, а основное назначение не статических методов (методов экземпляров) — реализация операций с типами данных. В клиентском коде могут присутствовать оба вида методов, но их легко различить: перед статическими методами указывается имя *класса* (по соглашению с большой буквы), а перед методами экземпляров — имя *объекта* (по соглашению с маленькой буквы). Эти различия приведены в табл. 1.2.1.

Таблица 1.2.1. Методы экземпляров и статические методы

	Метод экземпляра	Статический метод
Пример вызова	`head.increment()`	`Math.sqrt(2.0)`
В начале вызова находится	Имя объекта	Имя класса
Параметры	Ссылка на объект и аргумент(ы)	Аргумент(ы)
Основное назначение	Просмотр или изменение значения объекта	Вычисление возвращаемого значения

Использование объектов

Объявления дают нам имена переменных для объектов, которые можно использовать в коде не только для создания объектов и вызова методов экземпляров, но и точно так же, как имена переменных для целых чисел, вещественных чисел и других примитивных типов. Для разработки клиентского кода с использованием некоторого типа данных понадобится выполнить следующий действия.

- Объявить переменную этого типа, чтобы ссылаться на нужный объект.
- Использовать ключевое слово `new` для вызова конструктора, который создаст объект указанного типа.
- Использовать имя объекта для вызова методов экземпляров — в операторах или в составе выражений.

Например, класс `Flips`, приведенный в листинге 1.2.1, представляет собой клиент для типа данных `Counter`, который принимает из командной строки аргумент `T` и моделирует `T` подбрасываний монеты (заодно это клиент класса `StdRandom`). Кроме таких способов применения, переменные, связанные с объектами, можно использовать точно так же, как и переменные, связанные со значениями примитивных типов:

- в операторах присваивания;
- для передачи объектов в методы и возврата объектов из методов;
- для создания и использования массивов объектов.

Листинг 1.2.1. Клиент типа Counter, эмулирующий T подбрасываний монеты

```
public class Flips
{
   public static void main(String[] args)
   {
      int T = Integer.parseInt(args[0]);
      Counter heads = new Counter("орлов");
      Counter tails = new Counter("решек");
      for (int t = 0; t < T; t++)
         if (StdRandom.bernoulli(0.5))
              heads.increment();
         else tails.increment();
      StdOut.println(heads);
      StdOut.println(tails);
      int d = heads.tally() - tails.tally();
      StdOut.println("разница: " + Math.abs(d));
   }
}
```

```
% java Flips 10
5 орлов
5 решек
разница: 0
% java Flips 10
8 орлов
2 решек
разница: 6
% java Flips 1000000
499710 орлов
500290 решек
разница: 580
```

Чтобы четко понимать такое поведение, нужно мыслить и оперировать *ссылками*, а не значениями, и об этом мы сейчас поговорим более предметно.

Операторы присваивания

Оператор присваивания со ссылочным типом создает копию ссылки — не новый объект, а просто еще одну ссылку на существующий объект. Эта ситуация называется *созданием псевдонима*: обе переменные указывают на один и тот же объект. Эффект псевдонима несколько неожидан, т.к. он отличается от присваивания переменным примитивных типов. Не забывайте об этом различии.

Если x и y — переменные примитивного типа, то присваивание x = y копирует в переменную x значение y. Но у ссылочных типов копируется *ссылка*, а не значение. Псевдонимы являются источником многочисленных ошибок в Java-программах, как показано в следующем примере (см. также рис. 1.2.5):

```
Counter c1 = new Counter("единицы");
c1.increment();
Counter c2 = c1;
c2.increment();
StdOut.println(c1);
```

При типичной реализации метода `toString()` этот код выведет строку `"2 единицы"` — это может быть, а может и не быть тем, что вы ожидали, но уж точно выглядит неестественно. Такие ошибки часто встречаются в программах, авторы которых имеют мало опыта работы с объектами — это можете быть и вы, так что будьте внимательнее! Изменение состояния объекта влияет на весь код, где участвуют переменные-псевдонимы, указывающие на тот же объект. Мы привыкли, что две различные переменные примитивного типа не зависят друг от друга, но эта привычка не должна распространяться на переменные ссылочных типов.

```
Counter c1;
c1 = new Counter("единицы");
c1.increment();
Counter c2 = c1;
c2.increment();
```

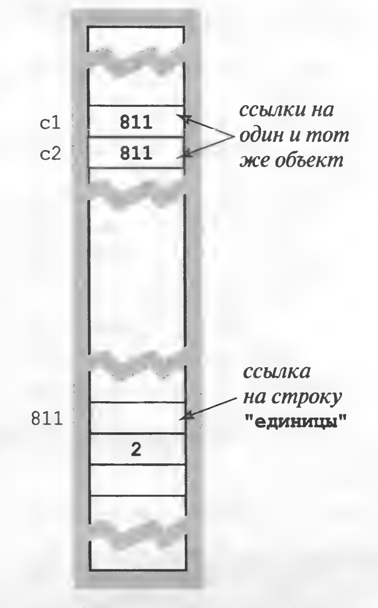


Рис. 1.2.5. Создание псевдонима

Объекты в качестве аргументов

Объекты можно передавать методам в качестве *аргументов*. Эта возможность обычно упрощает код клиента. Например, передавая в качестве аргумента объект `Counter`, мы, по сути, передаем сразу и название, и счетчик, хотя указываем лишь одну переменную. При вызове метода с аргументами-объектами получается так, как будто выполняются операторы присваивания со значениями параметров в правой части и именами аргументов в левой части. То есть Java передает из вызывающей программы в метод *копию* значения аргумента. Этот принцип называется *передачей по значению* (см. раздел 1.1, подраздел “Свойства методов”).

Важным следствием этого принципа является то, что метод не может изменить значение переменной из вызвавшего метода. Для примитивных типов тут все понятно (две переменные полностью независимы), но при использовании ссылочного типа в качестве аргумента метода создается псевдоним, так что будьте внимательны.

Другими словами, по значению передается *ссылка* — т.е. создается ее копия — но эта копия ссылки указывает на тот же *объект*.

Например, если передать ссылку на объект типа `Counter`, метод не сможет изменить исходную ссылку (сделать так, чтобы она указывала на другой объект), но он *может* изменить значение объекта — например, вызвать с помощью этой ссылки метод `increment()`.

Объекты в качестве возвращаемых значений

Естественно, объект можно использовать и в качестве значения, *возвращаемого* из метода. Метод может возвратить объект, переданный ему в качестве аргумента, как в листинге 1.2.2, или может создать новый объект и возвратить ссылку на него. Это важная возможность, т.к. Java-методам разрешено возвращать лишь одно значение: использование объектов позволяет, по сути, возвратить несколько значений.

Листинг 1.2.2. Пример статического метода с объектами в качестве аргументов и возвращаемого значения

```
public class FlipsMax
{
   public static Counter max(Counter x, Counter y)
   {
      if (x.tally() > y.tally()) return x;
      else                       return y;
   }
   public static void main(String[] args)
   {
      int T = Integer.parseInt(args[0]);
      Counter heads = new Counter("орлы");
      Counter tails = new Counter("решки");
      for (int t = 0; t < T; t++)
         if (StdRandom.bernoulli(0.5))
             heads.increment();
         else tails.increment();
      if (heads.tally() == tails.tally())
           StdOut.println("ничья");
      else StdOut.println(max(heads, tails) + " победили");
   }
}
```

```
% java FlipsMax 1000000
500281 решки победили
```

Массивы — это объекты

В Java любое значение непримитивного типа представляет собой объект. В частности, объектами являются массивы. Как и в случае строк, для определенных операций с массивами имеется специальная языковая поддержка: объявление, инициализация и индексирование. Как и в случае любого другого объекта, при передаче массива в метод либо использовании переменной типа массива в правой части оператора присваивания создается копия ссылки на массив, а не копия самого массива. Это соглашение удобно в типичных случаях использования массивов, когда метод может изменять элементы массива, переупорядочивать его значения — как, например, в случае метода `java.util.Arrays.sort()` или `shuffle()` из табл. 1.1.6.

Массивы объектов

Как мы уже убедились, элементы массивов могут быть любого типа: аргумент `args[]` в наших реализациях `main()` представляет собой массив объектов `String`. Создание массива объектов выполняется в два этапа:

- создается массив с указанием квадратных скобок для конструктора массива;
- создаются все объекты этого массива, для чего применяется стандартный конструктор.

Например, в листинге 1.2.3 моделируется бросание игральной кости, и объекты `Counter` используются для отслеживания количества выпадений каждого возможного значения. Массив объектов в Java представляет собой массив ссылок на объекты, а не массив

самих объектов. При большом размере объектов это удобно, поскольку перемещать в памяти нужно не их, а лишь ссылки. При малом размере эффективность может снизиться, т.к. каждый раз, когда понадобится информация, приходится переходить по ссылкам.

Листинг 1.2.3. Клиент типа `Counter`, моделирующий `T` бросаний игральной кости

```
public class Rolls
{
  public static void main(String[] args)
  {
    int T = Integer.parseInt(args[0]);
    int SIDES = 6;
    Counter[] rolls = new Counter[SIDES+1];
    for (int i = 1; i <= SIDES; i++)
      rolls[i] = new Counter("выпадений " + i);
    for (int t = 0; t < T; t++)
    {
      int result = StdRandom.uniform(1, SIDES+1);
      rolls[result].increment();
    }
    for (int i = 1; i <= SIDES; i++)
      StdOut.println(rolls[i]);
  }
}
```

```
% java Rolls 1000000
167308 выпадений 1
166540 выпадений 2
166087 выпадений 3
167051 выпадений 4
166422 выпадений 5
166592 выпадений 6
```

Эти принципы работы с объектами и написание кода, в котором применяется абстракция данных (определение и использование типов данных, когда значения типа данных хранятся в объектах), повсеместно называются *объектно-ориентированным программированием*. Вышеизложенные концепции являются основами объектно-ориентированного программирования, поэтому имеет смысл привести их краткую сводку.

Тип данных — это множество значений и множество операций, определенных для этих значений. Типы данных реализуются в независимых модулях Java-классов, а затем можно писать клиентские программы, в которых эти типы используются. *Объект* или *экземпляр* типа данных — это сущность, которая может принимать значение типа данных. Объекты характеризуются тремя ключевыми свойствами: *состояние*, *идентичность* и *поведение*. Реализация типа данных поддерживает клиенты типа данных следующим образом.

- Клиентский код может *создавать объекты* (идентифицировать) с помощью операции `new`, вызывающей конструктор, который создает объект, инициализирует его переменные экземпляров и возвращает ссылку на этот объект.
- Клиентский код может *обрабатывать значения типа данных* (управлять поведение объекта, возможно, с изменением его состояния): с помощью переменной, связан-

ной с объектом, вызывается метод экземпляра, который работает с переменными этого экземпляра объекта.

- Клиентский код может *работать с объектами* — создавать массивы объектов, передавать их в методы и возвращать из методов — точно так же, как и со значениями примитивных типов, только переменные содержат ссылки на значения, а не сами значения.

Эти возможности позволяют основать гибкий, современный и широко применяемый стиль программирования, который мы будем использовать для изучения алгоритмов в данной книге.

Примеры абстрактных типов данных

В языке Java имеются тысячи встроенных АТД, и мы определили много других АТД, которые облегчают изучение алгоритмов. Вообще-то любая написанная нами Java-программа представляет собой реализацию типа данных (или библиотеку статических методов). Для управления сложностью мы будем цитировать конкретные API-интерфейсы для каждого АТД, который будет использован в данной книге (это не очень много).

В этом разделе мы познакомимся с примерами нескольких типов данных и клиентского кода. В некоторых случаях мы приведем выдержки из API-интерфейсов, которые могут содержать десятки методов экземпляров. Эти API-интерфейсы будут приведены как примеры из реальной практики, для описания методов экземпляров, которые будут использоваться в данной книге, и чтобы подчеркнуть, что для применения АТД нет необходимости знать детали их реализаций.

Для справки типы данных, которые мы будем использовать и разрабатывать в настоящей книге, приведены на рис. 1.2.6. Они разбиты на несколько категорий.

- Стандартные системные АТД в пакете `java.lang.*`, которые можно использовать в любой Java-программе.
- АТД в Java-библиотеках `java.awt`, `java.net` и `java.io`, которые также можно использовать в любой Java-программе, но с помощью оператора `import`.
- Наши АТД ввода-вывода, которые позволяют работать с несколькими потоками ввода-вывода, аналогичными `StdIn` и `StdOut`.
- АТД работы с данными, основное назначение которых — облегчение организации и обработки данных с помощью инкапсуляции их представления. Ниже в данном разделе мы опишем несколько примеров для приложений в вычислительной геометрии и обработке информации, а позже используем их в качестве примеров в клиентском коде.
- АТД коллекций, основное назначение которых — облегчение работы с коллекциями однотипных данных. Типы `Bag`, `Stack` и `Queue` будут описаны в разделе 1.3, тип `PQ` — в главе 2, а типы `ST` и `SET` — в главах 3 и 5.
- АТД операций, которые мы будем использовать для анализа алгоритмов, как описано в разделах 1.4 и 1.5.
- АТД для алгоритмов обработки графов: АТД, ориентированные на данные, которые предназначены для инкапсуляции представления различных видов графов, и АТД, ориентированные на действия, которые предназначены для обеспечения спецификаций для алгоритмов обработки графов.

Стандартные системные типы Java из пакета `java.lang`	
`Integer`	*оболочка* `int`
`Double`	*оболочка* `double`
`String`	*индексированные* `char`
`StringBuilder`	*построитель строк*
Другие Java-типы	
`java.awt.Color`	*цвета*
`java.awt.Font`	*шрифты*
`java.net.URL`	*URL-адреса*
`java.io.File`	*файлы*
Наши стандартные типы ввода-вывода	
`In`	*поток ввода*
`Out`	*поток вывода*
`Draw`	*чертеж*
Типы работы с данными для примеров клиентов	
`Point2D`	*точка на плоскости*
`Interval1D`	*одномерный интервал*
`Interval2D`	*двухмерный интервал*
`Date`	*дата*
`Transaction`	*транзакция*
Типы для анализа алгоритмов	
`Counter`	*счетчик*
`Accumulator`	*накопитель*
`VisualAccumulator`	*визуальный вариант накопителя*
`Stopwatch`	*секундомер*
Типы коллекций	
`Stack`	*стек LIFO*
`Queue`	*очередь FIFO*
`Bag`	*контейнер*
`MinPQ MaxPQ`	*очередь с приоритетами*
`IndexMinPQ IndexMaxPQ`	*очередь с приоритетами (индексированная)*
`ST`	*таблица имен*
`SET`	*множество*
`StringST`	*таблица имен (строковые ключи)*
Типы графов, ориентированные на данные	
`Graph`	*граф*
`Digraph`	*ориентированный граф*
`Edge`	*ребро (с весом)*
`EdgeWeightedGraph`	*граф (взвешенный)*
`DirectedEdge`	*ребро (направленное с весом)*
`EdgeWeightedDigraph`	*граф (ориентированный взвешенный)*
Типы графов, ориентированные на действия	
`UF`	*динамическая связность*
`DepthFirstPaths`	*поиск пути в глубину*
`CC`	*связные компоненты*
`BreadthFirstPaths`	*поиск пути в ширину*
`DirectedDFS`	*поиск пути в глубину на диграфе*
`DirectedBFS`	*поиск пути в ширину на диграфе*
`TransitiveClosure`	*все пути*
`Topological`	*топологический порядок*
`DepthFirstOrder`	*порядок DFS*
`DirectedCycle`	*поиск циклов*
`SCC`	*сильные компоненты*
`MST`	*минимальное остовное дерево*
`SP`	*кратчайшие пути*

***Рис. 1.2.6.** Список АТД, используемых в книге*

Этот список не включает десятки типов, которые будут рассматриваться в упражнениях и в тексте книги. Кроме того, как сказано ниже в разделе “Другие реализации АТД”, мы часто будем отличать различные реализации АТД с помощью описательного префикса. Все вместе используемые нами АТД демонстрируют, что организация и понимание используемых типов данных — важный фактор в современном программировании.

Типичное приложение может использовать лишь 5–10 таких АТД. Основная цель разработки и организации АТД в этой книге — позволить программистам легко использовать относительно небольшое их множество для разработки клиентского кода.

Геометрические объекты

Естественный пример объектно-ориентированного программирования — проектирование типов данных для геометрических объектов. Например, API-интерфейсы, представленные на рис. 1.2.7, 1.2.8 и 1.2.9, определяют абстрактные типы данных для трех знакомых геометрических объектов: `Point2D` (точки на плоскости), `Interval1D` (интервалы на прямой) и `Interval2D` (двумерные интервалы на плоскости, т.е. прямоугольники, ориентированные вдоль осей координат).

public class **Point2D**		
	`Point2D(double x, double y)`	*создание точки*
`double`	`x()`	*координата x*
`double`	`y()`	*координата y*
`double`	`r()`	*радиус (полярные координаты)*
`double`	`theta()`	*угол (полярные координаты)*
`double`	`distTo(Point2D that)`	*евклидово расстояние от данной точки до* `that`
`void`	`draw()`	*вычерчивание точки в* `StdDraw`

Рис. 1.2.7. API-интерфейс точек на плоскости

public class **Interval1D**		
	`Interval1D(double lo, double hi)`	*создание интервала*
`double`	`length()`	*длина интервала*
`boolean`	`contains(double x)`	*содержит ли интервал точку x?*
`boolean`	`intersects(Interval1D that)`	*пересекается ли интервал с* `that`*?*
`void`	`draw()`	*вычерчивание интервала в* `StdDraw`

Рис. 1.2.8. API-интерфейс интервалов на линии

public class **Interval2D**		
	`Interval2D(Interval1D x, Interval1D y)`	*создание 2D-интервала*
`double`	`area()`	*площадь 2D-интервала*
`boolean`	`contains(Point p)`	*содержит ли 2D-интервал точку* `p`*?*
`boolean`	`intersects(Interval2D that)`	*пересекается ли 2D-интервал с* `that`*?*
`void`	`draw()`	*вычерчивание 2D-интервала в* `StdDraw`

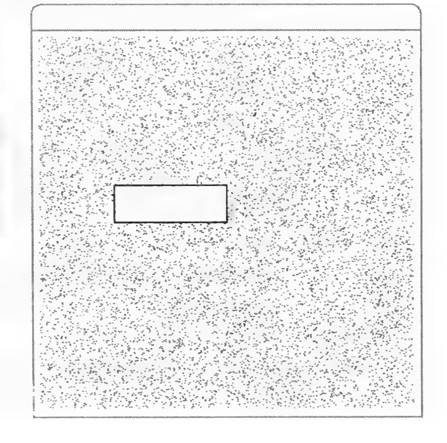
Рис. 1.2.9. API-интерфейс двумерных интервалов на плоскости

Как обычно, эти API-интерфейсы являются самодокументированными и облегчают понимание кода клиента, такого как приведенный в листинге 1.2.4: эта программа считывает из командной строки границы интервала `Interval2D` и целое число *T*, генерирует *T* случайных точек на единичном квадрате и подсчитывает количество точек, которые попали в интервал (оценка площади прямоугольника). Для наглядности клиент вычерчивает интервал и не попавшие в него точки. Это вычисление демонстрирует, как можно свести задачу вычисления площади и объема геометрических фигур к задаче определения, попала точка в фигуру или нет (менее сложная, хотя тоже нетривиальная задача). Конечно, можно определить API-интерфейсы и для других геометрических объектов: отрезков прямых линий, треугольников, многоугольников, окружностей и т.д., хотя реализация операций для них может оказаться трудной задачей. Несколько примеров приведены в упражнениях в конце данного раздела.

Листинг 1.2.4. Клиент тестирования для типа `Interval2D`

```
public static void main(String[] args)
{
   double xlo = Double.parseDouble(args[0]);
   double xhi = Double.parseDouble(args[1]);
   double ylo = Double.parseDouble(args[2]);
   double yhi = Double.parseDouble(args[3]);
   int T = Integer.parseInt(args[4]);
   Interval1D xint = new Interval1D(xlo, xhi);
   Interval1D yint = new Interval1D(ylo, yhi);
   Interval2D box = new Interval2D(xint, yint);
   box.draw();
   Counter c = new Counter("попаданий");
   for (int t = 0; t < T; t++)
   {
      double x = Math.random();
      double y = Math.random();
      Point2D p = new Point2D(x, y);
      if (box.contains(p)) c.increment();
      else                 p.draw();
   }
   StdOut.println(c);
   StdOut.println(box.area());
}
```

```
% java Interval2D .2 .5 .5 .6 10000
297 попаданий
.03
```



Программы, обрабатывающие геометрические объекты, широко применяются в вычислениях с моделями из обычного мира, в научных вычислениях, видеоиграх, фильмах и многих других приложениях. Разработка и изучение таких программ и приложений вылилась в целую область — *вычислительную геометрию*, которая изобилует примерами применения алгоритмов, рассматриваемых в книге. Сейчас мы хотим показать, что совсем не трудно определить абстрактные типы данных, которые непосредственно представляют собой геометрические абстракции и приводят к простому и ясному клиентскому коду. Эта идея демонстрируется в нескольких упражнениях в конце данного раздела и на сайте книги.

Обработка информации

Банк, обрабатывающий миллионы транзакций по кредитным картам, аналитическая компания, обрабатывающая миллиарды щелчков кнопкой мыши на веб-ссылках, научно-исследовательская группа, обрабатывающая миллионы замеров — в основе огромного множества приложений лежит обработка и организация информации. Абстрактные типы данных обеспечивают естественный механизм для организации информации. Например, API-интерфейс, представленный на рис. 1.2.10, предлагает типичный подход для некоторого коммерческого приложения.

public class **Date** implements Comparable<Date>		
	Date(int month, int day, int year)	*создание даты*
	Date(String date)	*создание даты (конструктор из строки)*
int	month()	*месяц*
int	day()	*день*
int	year()	*год*
String	**toString()**	*строковое представление*
boolean	**equals(Object that)**	*совпадает ли дата с* that?
int	**compareTo(Date that)**	*сравнение даты с* that
int	**hashCode()**	*хеш-код*

public class **Transaction** implements Comparable<Transaction>		
	Transaction(String who, Date when, double amount)	
	Transaction(String transaction)	*создание транзакции (конструктор из строки)*
String	who()	*имя клиента*
Date	when()	*дата*
double	amount()	*сумма*
String	**toString()**	*строковое представление*
boolean	**equals(Object that)**	*совпадает ли дата с* that?
int	**compareTo(Date that)**	*сравнение даты с* that
int	**hashCode()**	*хеш-код*

Рис. 1.2.10. *API-интерфейсы для коммерческих приложений (даты и транзакции)*

Идея состоит в том, чтобы определить типы данных, которые позволяют хранить данные в объектах, соответствующих явлениям и предметам реального мира. Дата состоит из дня, месяца и года, а транзакция — из клиента, даты и суммы. И это лишь два примера: мы могли бы определить типы данных, которые содержат подробную информацию о клиентах, моментах времени, местоположениях, товарах и услугах — и о чем угодно еще. Каждый тип данных состоит из конструкторов, которые создают объекты, содержащих данные, и методов для доступа к этим данным из клиентского кода. Для упрощения кода мы предоставляем по два конструктора для каждого типа: один представляет данные в соответствующем типе, а другой выбирает данные из строки (подробнее см. в упражнении 1.2.19). Как обычно, клиентскому коду совершенно не обязательно знать представление данных. Чаще всего причиной такой организации данных является желание работать с данными, связанными с объектом, как с единым целым: можно использовать массивы значений `Transaction`, применять значения `Date` в качестве аргумента или возвращаемого значения метода и т.д.

Главное, что позволяют такие типы данных — инкапсулировать данные и одновременно разрабатывать клиентский код, который не зависит от представления этих данных. Мы не будем уделять много внимания подобной организации информации, лишь отметим, что наряду с унаследованными методами `toString()`, `compareTo()`, `equals()` и `hashCode()` это позволяет воспользоваться реализациями алгоритмов, которые могут работать *с любыми типами данных*. Унаследованные методы будут рассмотрены ниже, в разделе “Наследование интерфейса”. Например, уже было упомянуто соглашение, которое позволяет клиентам выводить строковое представление любого значения, если в типе данных имеется реализация метода `toString()`. Мы рассмотрим соглашения, соответствующие другим наследуемым методам, в разделах 1.3, 2.5, 3.4 и 3.5, на примере типов `Date` и `Transaction`.

В разделе 1.3 приведены классические примеры типов данных и описан механизм языка Java, который называется *параметризованными типами* или *обобщениями* и который также использует эти соглашения. Главы 2 и 3 также посвящены использованию обобщенных типов и наследуемых методов для разработки реализаций алгоритмов сортировки и поиска, эффективных для любых типов данных.

При наличии данных различных типов, которые логически взаимосвязаны, имеет смысл потратить усилия на определение АТД, как в приведенных примерах. Это помогает организовать данные, значительно упрощает код клиента в типичных приложениях и является важным шагом на пути абстракции данных.

Строки

Java-тип `String` — важный и нужный АТД. Он представляет собой индексированную последовательность значений типа `char`. У этого типа имеются десятки методов экземпляров, и некоторые из них перечислены на рис. 1.2.11.

Значения `String` похожи на массивы символов, но не эквивалентны им. Для доступа к отдельным символам массива в Java имеется встроенный синтаксис, а в типе `String` предусмотрены методы экземпляров для индексированного доступа, получения длины строки и многих других операций (рис. 1.2.12). Зато тип `String` пользуется специальной языковой поддержкой для инициализации и конкатенации: вместо создания и инициализации строки с помощью конструктора можно применять строковый литерал, а вместо вызова метода `concat()` использовать операцию +. Нет необходимости рассматривать здесь детали реализации, хотя, как мы убедимся в главе 5, понимание характеристик производительности некоторых методов важно при разработке алгоритмов обработки строк.

public class **String**		
	String()	*создание пустой строки*
int	length()	*длина строки*
int	charAt(int i)	*i-й символ*
int	indexOf(String p)	*первое вхождение p (-1, если нет)*
int	indexOf(String p, int i)	*первое вхождение p после i (-1, если нет)*
String	concat(String t)	*добавление строки t*
String	substring(int i, int j)	*подстрока (символы с i до j-1)*
String[]	split(String delim)	*строки между вхождениями delim*
int	**compareTo(String t)**	*сравнение строк*
boolean	**equals(String t)**	*совпадает ли строка с t?*
int	**hashCode()**	*хеш-код*

Рис. 1.2.11. *API-интерфейс строк в Java (часть списка методов)*

```
String a = "now is ";
String b = "the time ";
String c = "to"
```

Вызов	**Значение**
a.length()	7
a.charAt(4)	i
a.concat(c)	"now is to"
a.indexOf("is")	4
a.substring(2, 5)	"w i"
a.split(" ")[0]	"now"
a.split(" ")[1]	"is"
b.equals(c)	false

Рис. 1.2.12. *Примеры строковых операций*

Почему вместо массивов символов обычно используются значения типа String? Ответ на этот вопрос характерен для всех АТД: *чтобы сделать клиентский код проще и понятнее*. Тип String позволяет писать ясный и простой код, в котором применяются многочисленные удобные методы экземпляров, без необходимости знать, как конкретно представляются строки — см. табл. 1.2.2. Даже такой краткий список содержит мощные операции, которые требуют нетривиальных алгоритмов, таких как рассматриваемые в главе 5. Например, аргумент метода split() может быть *регулярным выражением* (см. раздел 5.4): пример split("\\s+") содержит аргумент "\\s+", который означает "один или более символов табуляции, пробела, новой строки или возврата строки".

Таблица 1.2.2. Типичные примеры кода для обработки строк

Задача	Реализация
Является ли строка палиндромом?	`public static boolean isPalindrome(String s)` `{` `  int N = s.length();` `  for (int i = 0; i < N/2; i++)` `    if (s.charAt(i) != s.charAt(N-1-i))` `      return false;` `  return true;` `}`
Извлечь имя и расширение файла из аргумента командной строки	`String s = args[0];` `int dot = s.indexOf(".");` `String base = s.substring(0, dot);` `String extension =` `  s.substring(dot + 1, s.length());`
Вывести в стандартный вывод все строки, содержащие строку, которая указана в командной строке	`String query = args[0];` `while (!StdIn.isEmpty())` `{` `  String s = StdIn.readLine();` `  if (s.contains(query)) StdOut.println(s);` `}`
Создать массив строк из StdIn, разделенных пробельными символами	`String input = StdIn.readAll();` `String[] words = input.split("\\s+");`
Проверить, упорядочен ли массив строк по алфавиту	`public boolean isSorted(String[] a)` `{` `  for (int i = 1; i < a.length; i++)` `  {` `    if (a[i-1].compareTo(a[i]) > 0)` `      return false;` `  }` `  return true;` `}`

Еще раз о вводе и выводе

Недостатком стандартных библиотек `StdIn`, `StdOut` и `StdDraw`, описанных в разделе 1.1, является то, что в любой программе они позволяют работать только с одним файлом ввода, одним файлом вывода и одним чертежом. Объектно-ориентированное программирование позволяет определить аналогичные механизмы для работы с *несколькими* потоками ввода, вывода и вычерчивания. А именно, наша стандартная библиотека содержит типы данных `In`, `Out` и `Draw` с API-интерфейсами, приведенными на рис. 1.2.13–1.2.15. Если конструкторы классов `In` и `Out` вызываются с аргументом типа `String`, они сначала пытаются найти в текущем каталоге файл с таким именем. Если такого файла нет, конструктор считает, что это имя веб-сайта, и он пытается подключиться к этому веб-сайту (если сайт не найден, то возникнет исключение времени выполнения). В любом случае указанный файл или сайт становится источником или приемником созданного объекта потока, а методы `read*()` и `print*()` будут оперировать с этим файлом или сайтом. (При вызове конструкторов без аргументов будут получены стандартные потоки.) Это позволяет одной программе работать с несколькими файлами и чертежами. Такие объекты можно присваивать переменным, создавать из них масси-

вы и обрабатывать так же, как и объекты любого другого типа. Программа `Cat`, приведенная в листинге 1.2.5, представляет собой клиент классов `In` и `Out` и использует несколько входных потоков для конкатенации содержимого нескольких входных файла в один выходной файл. Классы `In` и `Out` содержат также статические методы для чтения из файлов в массив значений типов `int`, `double` или `String` (см. листинг 1.3.2 и упражнение 1.2.15).

public class **In**		
	`In()`	*создание потока ввода для стандартного ввода*
	`In(String name)`	*создание потока ввода для файла или веб-сайта*
`boolean`	`isEmpty()`	*`true`, если данных больше нет, иначе `false`*
`int`	`readInt()`	*чтение значения типа `int`*
`double`	`readDouble()`	*чтение значения типа `double`*
	`...`	
`void`	`close()`	*закрытие потока ввода*

Примечание: все операции, поддерживаемые в `StdIn`, поддерживаются и в объектах `In`.

***Рис. 1.2.13.** API-интерфейс для потоков ввода*

public class **Out**		
	`Out()`	*создание потока вывода для стандартного вывода*
	`Out(String name)`	*создание потока вывода в файл*
`void`	`print(String s)`	*добавление строки **s** в поток вывода*
`void`	`println(String s)`	*добавление строки **s** и новой строки в поток вывода*
`void`	`println()`	*добавление символа новой строки в поток вывода*
`void`	`printf(String f, ...)`	*вывод с форматированием в поток вывода*
`void`	`close()`	*закрытие потока вывода*

Примечание: все операции, поддерживаемые в `StdOut`, поддерживаются и в объектах `Out`.

***Рис. 1.2.14.** API-интерфейс для потоков вывода*

public class **Draw**	
	`Draw()`
`void`	`line(double x0, double y0, double x1, double y1)`
`void`	`point(double x, double y)`
	`...`

Примечание: все операции, поддерживаемые в `StdDraw`, поддерживаются и в объектах `Draw`.

***Рис. 1.2.15.** API-интерфейс для чертежей*

Листинг 1.2.5. Пример клиента для классов In и Out

```
public class  Cat
{
  public static void main(String[] args)
  {  // Копирование входных файлов в out (последний аргумент).
     Out out = new Out(args[args.length-1]);
     for (int i = 0; i < args.length - 1; i++)
     {  // Копирование в out входного файла с именем из i-го аргумента.
        In in = new In(args[i]);
        String s = in.readAll();
        out.println(s);
        in.close();
     }
     out.close();
  }
}
```

```
% more in1.txt
Это -
% more in2.txt
небольшой
тест.
% java Cat in1.txt in2.txt out.txt
% more out.txt
Это -
небольшой
тест.
```

Реализация абстрактного типа данных

Как и в случае библиотек статических методов, АТД реализуется с помощью Java-класса, код которого помещается в файл с именем, совпадающим с именем класса, и с расширением `.java`. Первые операторы в таком файле объявляют *переменные экземпляров*, которые определяют значения типа данных. За ними следуют *конструктор* и *методы экземпляров*, реализующие операции над значениями типа данных.

Методы экземпляров могут быть *общедоступными* (описанными в API-интерфейсе) или *приватными* (применяются для организации вычислений и не доступны клиентам). Определение типа данных может содержать несколько конструкторов и может также включать определения статических методов. В частности, клиент `main()` обычно удобен для тестирования и отладки. В качестве первого примера мы рассмотрим реализацию АТД `Counter`, который был определен на рис. 1.2.1. Полная реализация с комментариями приведена на рис. 1.2.16 — для обсуждения частей, составляющих класс. Каждая реализация АТД, которую мы будем разрабатывать, имеет те же базовые компоненты, что и в этом простом примере.

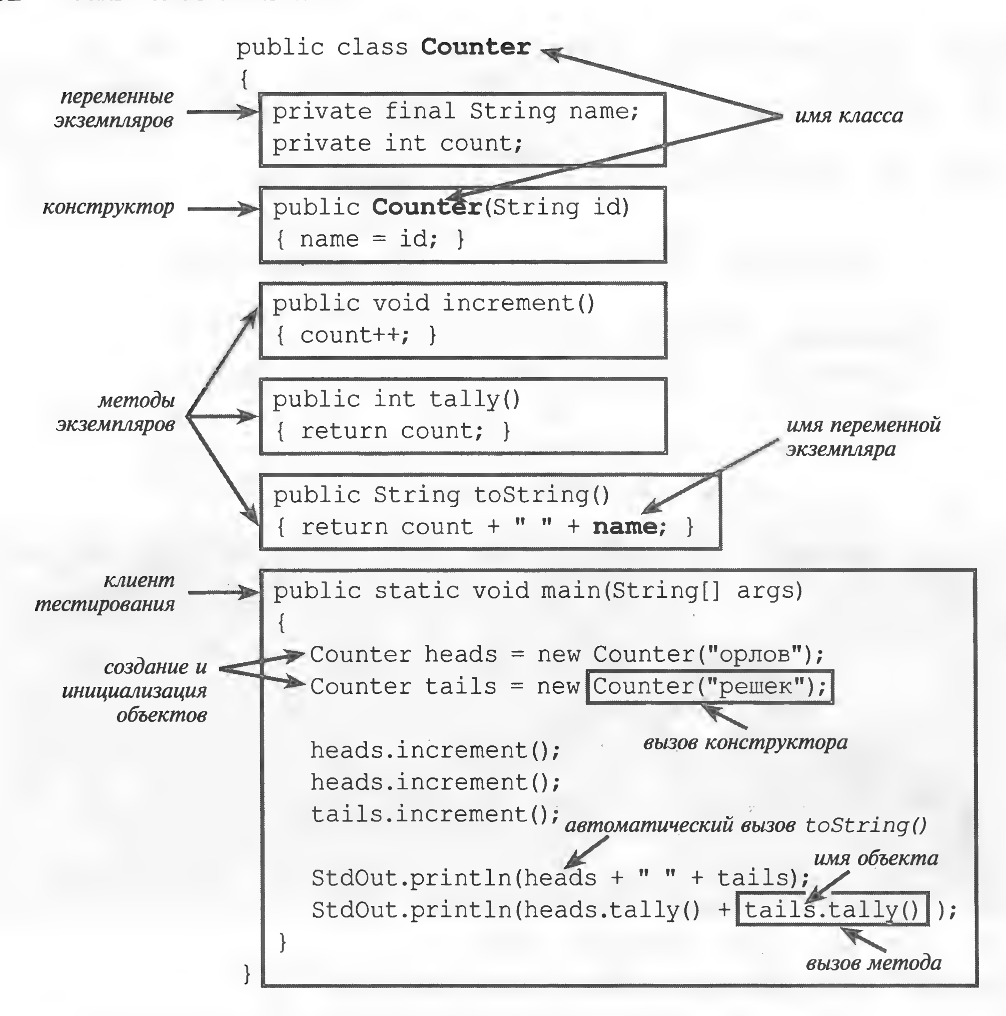


Рис. 1.2.16. Структура класса с определением типа данных

Методы экземпляров

Чтобы определять значения типа данных (*состояние* каждого объекта), мы объявляем *переменные экземпляров* — примерно так же, как объявляем локальные переменные. Однако имеется и существенное различие между переменными экземпляров и локальными переменными в каком-то статическом методе или блоке: в любой конкретный момент времени существует лишь *одно* значение, соответствующее каждой локальной переменной, но *несколько* значений, соответствующих каждой переменной экземпляра (по одному в каждом объекте, который является экземпляром типа данных). Это не приводит к неоднозначности, поскольку при любом обращении к переменной экземпляра указывается имя объекта — мы обращаемся к значению именно этого объекта. Кроме того, каждое объявление квалифицируется *модификатором видимости*. В реализациях АТД применяются ключевые слова `private` (представление АТД должно быть скрыто от клиента) и `final` (значение не должно изменяться после инициализации, рис. 1.2.17).

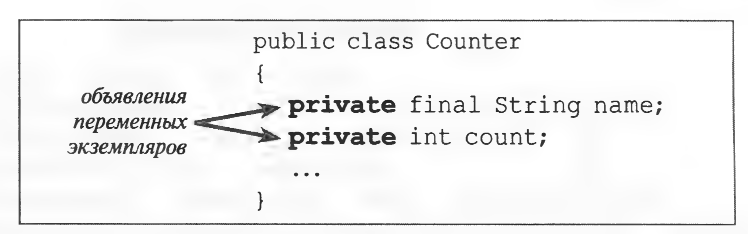


Рис. 1.2.17. Переменные экземпляров в АТД объявлены как `private`

Класс `Counter` содержит две переменные экземпляров: `name` типа `String` и `count` типа `int`. Если использовать переменные экземпляров вида `public` (допустимые в Java), то по определению тип данных не будет абстрактным, поэтому мы не будем делать этого.

Конструкторы

У каждого Java-класса имеется хотя бы один *конструктор*, который обеспечивает *идентичность* объекта. Конструктор похож на статический метод, но он может обращаться непосредственно к переменным экземпляров и не возвращает значения (рис. 1.2.18). В общем случае конструктор предназначен для инициализации переменных экземпляров. Каждый конструктор создает объект и предоставляет клиенту ссылку на этот объект. Имя конструктора всегда совпадает с именем класса. Как и другие методы, его можно перегрузить и иметь несколько конструкторов с различными сигнатурами. Если не определен никакой другой конструктор, неявно подразумевается стандартный конструктор без аргументов, который инициализирует переменные экземпляров значениями по умолчанию. Значениями по умолчанию для переменных экземпляров являются: 0 для примитивных числовых типов, `false` для логических и `null` для ссылочных. Эти значения можно изменить с помощью инициализирующих объявлений для переменных экземпляров. Java автоматически вызывает конструктор, когда клиентская программа использует операцию `new`. Перегруженные конструкторы обычно используются для инициализации переменных экземпляров значениями, которые предоставлены клиентом и отличны от значений по умолчанию. Например, в классе `Counter` имеется конструктор с одним аргументом, который инициализирует переменную экземпляра `name` значением из аргумента (а переменная экземпляра `count` инициализируется стандартным значением 0).

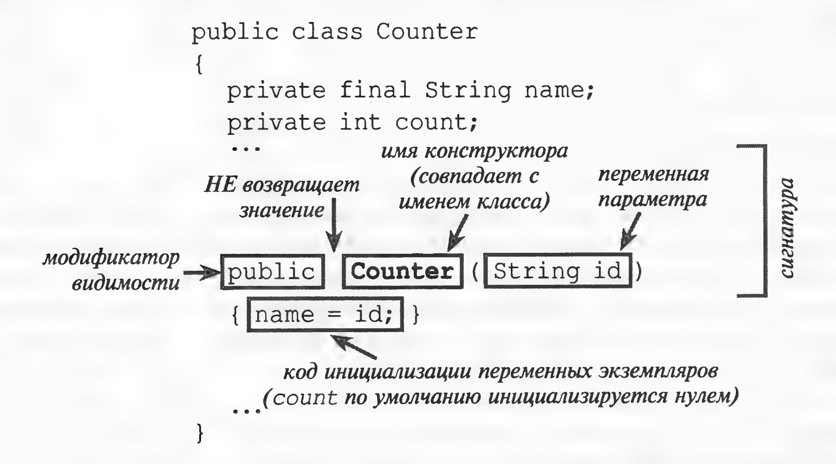


Рис. 1.2.18. Структура конструктора

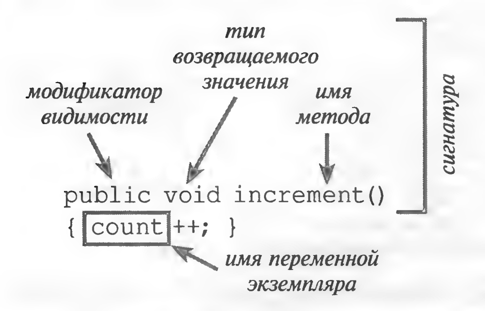


Рис. 1.2.19. Структура метода экземпляра

Методы экземпляров

Для реализации методов типа данных (*поведение* каждого объекта) мы реализуем *методы экземпляров* с кодом, который ничем не отличается от кода, рассмотренного в разделе 1.1 для реализации статических методов (функций). Каждый метод экземпляра имеет тип возвращаемого значения, *сигнатуру* (которая задает его имя и типы и имена переменных его параметров) и *тело* (состоит из последовательности операторов и включает оператор *возврата*, который предоставляет значение для возврата клиенту) (рис. 1.2.19). Когда клиент вызывает метод, значения параметров (если они есть) инициализируются значениями из клиента, операторы метода выполняется до вычисления возвращаемого значения, и значение возвращается клиенту — общий эффект таков, будто вызов метода в клиенте заменяется этим значением. Все это полностью совпадает со статическими методами, но есть и одно существенное отличие от них: такие методы *могут обращаться к переменным экземпляров и обрабатывать их*. Как указать, переменные экземпляров какого объекта нужно использовать? Если немного подумать, нетрудно догадаться о логичном ответе: ссылка на переменную в методе экземпляра ссылается на значение *для объекта, который был использован для вызова метода*. При вызове `heads.increment()` код в методе `increment()` ссылается на переменные экземпляров для объекта `heads`. То есть объектно-ориентированное программирование добавляет один крайне важный дополнительный способ для использования переменных в Java-программе:

- для вызова метода экземпляра, который работает со значениями объекта.

Отличие от работы только со статическими методами лежит в семантике (см. "Вопросы и ответы"), однако оно кардинально изменило способ мышления современных программистов в отношении разработки кода во многих ситуациях. И, как мы увидим в дальнейшем, оно тесно взаимосвязано с изучением алгоритмов и структур данных.

Область видимости

В общем Java-код, который мы будем применять для реализации методов экземпляров, задействует *три* вида переменных:

- переменные параметров;
- локальные переменные;
- *переменные экземпляров*.

Первые два из них такие же, как и для статических методов: переменные параметров определяются в сигнатуре метода и инициализируются клиентскими значениями при вызове метода, а локальные переменные объявляются и инициализируются в теле метода. Область действия переменных параметров — весь метод; область действия локальных переменных — следующие за объявлением операторы блока, в котором они определены. Переменные экземпляров кардинально отличаются от них: они содержат значения типа данных для объектов класса, а их область действия — весь класс (во избежание возможной неоднозначности можно использовать префикс `this`, обозначающий переменную экземпляра) (рис. 1.2.20). Четкое понимание различий между этими тремя видами переменных в методах экземпляров является залогом успеха в объектно-ориентированном программировании.

```
public class Example
{
   private int var;  ← переменная экземпляра
   ...
   private void method1()
   {
локальная переменная → int var;      указывает на локальную переменную,
      ...  var  ←                     А НЕ на переменную экземпляра
      ...  this.var  ← указывает на переменную экземпляра
   }
   private void method2()
   {
      ...  var      ...
   }        ↖ указывает на переменную экземпляра
   ...
}
```

Рис. 1.2.20. Область действия переменных экземпляров и локальных переменных в методе экземпляра

API-интерфейс, клиенты и реализации

Это базовые компоненты, в которых необходимо разобраться, чтобы создавать и использовать абстрактные типы данных в Java. Все реализации АТД, которые мы будем рассматривать, будут оформлены в виде Java-классов с приватными переменными экземпляров, конструкторами, методами экземпляров и клиентами. Для полного понимания типа данных потребуется API-интерфейс, код типичного клиента и реализация — как для класса `Counter` на рис. 1.2.21. Чтобы подчеркнуть это разделение клиента и реализации, мы обычно оформляем каждый клиент в виде отдельного класса, содержащего статический метод `main()`, и резервируем метод `main()` клиента тестирования в определении типа данных для минимальной проверки работоспособности модуля (каждый метод экземпляра вызывается хотя бы раз). При разработке каждого типа данных мы будем выполнять одни и те же шаги. Мы будем не рассуждать, какое действие следует выполнить следующим для достижения нашей цели (как это было раньше при первоначальном изучении программирования), а вначале обдумывать, что нужно клиенту, затем выражать эти потребности в виде АТД с помощью перечисленных ниже шагов.

- Формулировка API-интерфейса. Назначение API-интерфейса — *отделять клиенты от реализаций* с помощью модульного программирования. При записи API-интерфейса должны быть достигнуты две цели. Во-первых, нам нужен ясный и корректный клиентский код. Прежде чем завершить формулировку API-интерфейса, очень полезно хотя бы частично написать код клиента, чтобы быть уверенным, чтобы описанные для типа данных операции — именно те, которые нужны клиенту. Во-вторых, необходимо иметь возможность реализовать эти операции, ведь бессмысленно описывать операции, если непонятно, как их реализовывать.
- Реализация Java-класса, которая удовлетворяет спецификациям API-интерфейса. Вначале нужно выбрать переменные экземпляров, а затем написать конструкторы и методы экземпляров.
- Разработка нескольких клиентов тестирования для проверки проектировочных решений, принятых на первых двух шагах.

API-интерфейс

public class **Counter**		
	`Counter(String id)`	*создание счетчика с именем* `id`
`void`	`increment()`	*увеличение счетчика*
`int`	`tally()`	*количество увеличений с момента создания*
`String`	`toString()`	*строковое представление*

Типичный клиент

```
public class Flips
{
   public static void main(String[] args)
   {
      int T = Integer.parseInt(args[0]);
      Counter heads = new Counter("орлов");
      Counter tails = new Counter("решек");
      for (int t = 0; t < T; t++)
         if (StdRandom.bernoulli(0.5))
              heads.increment();
         else tails.increment();
      StdOut.println(heads);
      StdOut.println(tails);
      int d = heads.tally() - tails.tally();
      StdOut.println("разница: " + Math.abs(d));
   }
}
```

Реализация

```
public class Counter
{
   private final String name;
   private int count;
   public Counter(String id)
   { name = id; }
   public void increment()
   { count++; }
   public int tally()
   { return count; }
   public String toString()
   { return count + " " + name; }
}
```

Применение

```
% java Flips 1000000
500172 орлов
499828 решек
разница: 344
```

Рис. 1.2.21. Абстрактный тип данных для простого счетчика

Какие операции нужно выполнять клиенту, и какие значения типов данных могут лучше всего обеспечить выполнение этих операций? Ответы на эти принципиальные вопросы лежат в основе всех разрабатываемых нами реализаций.

Другие реализации АТД

Как и в случае любой концепции программирования, лучший способ уяснить мощь и удобство АТД — это тщательное рассмотрение дополнительных примеров и реализаций. У вас будет множество возможностей для этого, т.к. большая часть книги посвящена реализациям АТД, но несколько упрощенных примеров помогут заложить фундамент для последующей работы.

Дата

На рис. 1.2.22 показаны две реализации АТД `Date`, API-интерфейс для которого был приведен на рис. 1.2.10. Чтобы не загромождать рисунок, мы не включили конструктор из строки (который описан в упражнении 1.2.19) и унаследованные методы `equals()` (см. листинг 1.2.5), `compareTo()` (см. листинг 2.1.3) и `hashCode()` (см. упражнение 3.4.22). Примитивная (первая) реализация хранит день, месяц и год в переменных экземпляров, и методы экземпляров могут просто возвращать соответствующее значение; более экономная по отношению к памяти (вторая) реализация использует для представления даты лишь одно значение `int`: одно число в смешанной системе счисления представляет дату с днем d, месяцем m и годом y в виде $512y + 32m + d$. Клиент может обнаружить разницу между реализациями, к примеру, с помощью нарушений неявных предположений: корректность второй реализации зависит от того, что день может принимать значения от 0 до 31, месяц — от 0 до 15, а год должен быть положительным. (Вообще говоря, обе реализации должны проверять, что номера месяцев находятся в диапазоне 1–12, дни — от 1 до 31, а даты вроде 31 июня или 29 февраля 2009 г. некорректны — хотя это и требует дополнительных затрат.) Данный пример иллюстрирует идею, что в API-интерфейсе редко *полностью* определяются все требования к реализации (в данной книге мы стараемся выполнить это требование как можно полнее). Еще один способ, которым клиент может обнаружить разницу — это *производительность*: вторая реализация экономит память, необходимую для хранения значений данных, за счет дополнительного времени, необходимого для предоставления их клиенту в требуемом виде (нужно выполнить несколько арифметических операций). Такие вариации встречаются повсеместно: одному клиенту может быть более удобна одна реализация, а другому другая, а нам необходимо удовлетворить все требования. В настоящей книге постоянно рассматривается тема требований к памяти и времени выполнения для различных реализаций и их применимость для различных клиентов. Одним из основных преимуществ использования абстракции данных в наших реализациях является то, что обычно можно переключиться с одной реализации на другую, *совершенно не изменяя код клиента*.

Работа с несколькими реализациями

Наличие нескольких реализаций одного и того же API-интерфейса может привести к проблемам в сопровождении и именовании. В некоторых случаях нужно просто заменить старую реализацию новой, улучшенной. В других нужны обе реализации — одна удобнее для одних клиентов, другая для других. Вообще-то основная цель этой книги — тщательный анализ нескольких реализаций каждого из ряда фундаментальных АТД, обычно с различными характеристиками производительности.

4 Зак. 3239

API-интерфейс

public class **Date**		
	Date(int month, int day, int year)	*создание даты*
int	month()	*месяц*
int	day()	*день*
int	year()	*год*
String	toString()	*строковое представление*

Клиент тестирования

```
public static void main(String[] args)
{
   int m = Integer.parseInt(args[0]);
   int d = Integer.parseInt(args[1]);
   int y = Integer.parseInt(args[2]);
   Date date = new Date(m, d, y);
   StdOut.println(date);}
```

Применение

```
% java Date 12 31 1999
12/31/1999
```

Реализация

```
public class Date
{
   private final int month;
   private final int day;
   private final int year;
   public Date(int m, int d, int y)
   {  month = m; day = d; year = y; }
   public int month()
   {  return month;  }
   public int day()
   {  return day;  }
   public int year()
   {  return day;  }
   public String toString()
   {  return month() + "/" + day()
                    + "/" + year();  }
}
```

Другая реализация

```
public class Date
{
   private final int value;
   public Date(int m, int d, int y)
   { value = y*512 + m*32 + d; }
   public int month()
   { return (value / 32) % 16; }
   public int day()
   { return value % 32; }
   public int year()
   { return value / 512; }
   public String toString()
   {  return month() + "/" + day()
                 + "/" + year();  }
}
```

Рис. 1.2.22. Абстрактный тип данных для инкапсуляции дат с двумя реализациями

В книге мы часто будем сравнивать производительность одного клиента, использующего две различные реализации одного и того же API-интерфейса. По этой причине мы обычно придерживаемся неформальных соглашений по именованию, особенности которых описаны ниже.

- Различные реализации одного и того же API-интерфейса идентифицируются с помощью осмысленного префикса. Например, реализации типа `Date` на рис. 1.2.22 можно было бы назвать `BasicDate` и `SmallDate`, и можно было бы разработать реализацию `SmartDate`, которая проверяла бы правильность дат.
- Поддерживается эталонная реализация без префикса, которая выбирает способ работы, подходящий для большинства клиентов. То есть большинство клиентов должны использовать просто тип `Date`.

В крупных системах такое решение не идеально, поскольку может потребовать изменения кода клиента. Например, если бы мы взялись за новую реализацию `ExtraSmallDate`, то пришлось бы либо изменить код клиента, либо сделать эту реализацию эталонной, чтобы ее использовали все клиенты. В Java имеются различные дополнительные языковые механизмы для применения нескольких реализаций без необходимости изменять код клиента, но мы будем использовать их нечасто, т.к. это сложно и не очень естественно даже для экспертов — в особенности в сочетании с другими дополнительными языковыми возможностями, которые нам нравятся (обобщенные типы и итераторы). Эти вопросы важны — например, их игнорирование привело к знаменитой *проблеме Y2K* при смене тысячелетия, поскольку многие программы пользовались собственными реализациями абстракции даты, которые не учитывали первые две цифры года. Однако подробное рассмотрение таких вопросов может увести нас далеко от изучения алгоритмов.

Накопитель

API-интерфейс *накопителя*, приведенный на рис. 1.2.23, определяет абстрактный тип данных, который позволяет клиентам постоянно иметь текущее среднее значений данных. Этот тип данных часто используется в настоящей книге — к примеру, для обработки экспериментальных результатов (см. раздел 1.4). Реализация не представляет труда: в ней применяется счетчик в переменной экземпляра типа `int` для хранения количества введенных значений данных и переменная экземпляра `double` для хранения суммы этих значений; для вычисления среднего значения сумма делится на количество. Обратите внимание, что реализация не хранит сами значения данных: она может работать с огромным количеством замеров (даже на устройстве, которое не может содержать так много), а в большой системе можно использовать огромное количество таких накопителей. Эта особенность не очевидна и может быть специально пояснена в API-интерфейсе, т.к. реализация, хранящая в памяти все значения, может привести к исчерпанию памяти.

Визуальный накопитель

Реализация *визуального накопителя*, приведенная на рис. 1.2.25, расширяет тип `Accumulator` для получения полезного побочного эффекта: она выводит на `StdDraw` все данные (белым цветом) и текущее среднее значение (черным цветом), как показано на рис. 1.2.24. Это проще всего сделать, добавив конструктор, который принимает в качестве аргументов количество выводимых точек и максимально возможное значение для масштабирования чертежа.

API-интерфейс

public class **Accumulator**		
	`Accumulator()`	*создание накопителя*
`void`	`addDataValue(double val)`	*добавление нового значения данных*
`double`	`mean()`	*среднее всех значений данных*
`String`	`toString()`	*строковое представление*

Типичный клиент

```
public class TestAccumulator
{
   public static void main(String[] args)
   {
      int T = Integer.parseInt(args[0]);
      Accumulator a = new Accumulator();
      for (int t = 0; t < T; t++)
         a.addDataValue(StdRandom.random());
      StdOut.println(a);
   }
}
```

Применение

```
% java TestAccumulator 1000
Среднее (1000 значений): 0.51829

% java TestAccumulator 1000000
Среднее (1000000 значений): 0.49948

% java TestAccumulator 1000000
Среднее (1000000 значений): 0.50014
```

Реализация

```
public class  Accumulator
{
   private double total;
   private int N;
   public void addDataValue(double val)
   {
      N++;
      total += val;
   }
   public double mean()
   {  return total/N;  }
   public String toString()
   { return "Среднее (" + N + " значений): "
              + String.format("%7.5f", mean()); }
}
```

Рис. 1.2.23. Абстрактный тип данных для накопления значений данных

Высота N-й черной точки слева равна среднему значению высот всех N белых точек слева от нее

Высота белой точки представляет собой одно из значений данных

Рис. 1.2.24. *Принцип работы визуального накопителя*

Формально класс VisualAccumulator не является реализацией API-интерфейса Accumulator: его конструктор имеет другую сигнатуру и вызывает описанный выше побочный эффект. В общем случае мы стараемся полностью определять API-интерфейсы, и неохотно вносим *любые* изменения в API-интерфейс после его определения, поскольку это может потребовать изменений в неизвестном объеме кода клиентов (и реализации). Однако добавление конструктора для получения дополнительных возможностей иногда можно оправдать, т.к. при этом в клиентском коде изменяется та строка, которая все равно изменяется при смене имени класса. Если в нашем примере мы разработали клиент, который использует тип Accumulator и, возможно, многократно вызывает методы addDataValue() и mean(), то можно воспользоваться преимуществами типа VisualAccumulator, изменив лишь одну строку в коде клиента.

Проектирование типа данных

Абстрактный тип данных — это тип данных, представление которого скрыто от клиента. Этот принцип очень серьезно повлиял на современное программирование. Рассмотренные нами различные примеры дают понятие о дополнительных характеристиках АТД и их реализациях в виде Java-классов. Многие из этих тем слабо связаны с изучением алгоритмов, поэтому при желании вы можете пропустить данный раздел и вернуться к нему позднее, в контексте задач конкретных реализаций. Мы хотим собрать в одном месте важную информацию, которая относится к построению типов — как справочник и основу для реализаций во всей книге.

Инкапсуляция

Объектно-ориентированное программирование характерно тем, что оно позволяет *инкапсулировать* типы данных внутри их реализаций и разграничить разработку реализаций клиентов и типов данных. Инкапсуляция делает возможным модульное программирование, а оно позволяет:

- независимо разрабатывать код клиента и реализации;
- подставлять улучшенные реализации без влияния на клиенты;
- поддерживать еще не написанные программы (API-интерфейс является руководством для любого будущего клиента).

API-интерфейс

public class **VisualAccumulator**		
	VisualAccumulator(int trials, double max)	
void	addDataValue(double val)	*добавление нового значения данных*
double	mean()	*среднее всех значений данных*
String	toString()	*строковое представление*

Типичный клиент

```
public class TestVisualAccumulator
{
   public static void main(String[] args)
   {
      int T = Integer.parseInt(args[0]);
      VisualAccumulator a = new VisualAccumulator(T, 1.0);
      for (int t = 0; t < T; t++)
         a.addDataValue(StdRandom.random());
      StdOut.println(a);
   }
}
```

Применение

```
% java TestVisualAccumulator 2000
Среднее (2000 значений): 0.509789
```

Реализация

```
public class VisualAccumulator
{
   private double total;
   private int N;
   public VisualAccumulator(int trials, double max)
   {
      StdDraw.setXscale(0, trials);
      StdDraw.setYscale(0, max);
      StdDraw.setPenRadius(.005);
   }
   public void addDataValue(double val)
   {
      N++;
      total += val;
      StdDraw.setPenColor(StdDraw.DARK_GRAY);
      StdDraw.point(N, val);
      StdDraw.setPenColor(StdDraw.RED);
      StdDraw.point(N, total/N);
   }
   public double mean()
   public String toString()
   // Такие же, как в Accumulator.
}
```

Рис. 1.2.25. Абстрактный тип данных для накопления значений данных (визуальная версия)

Инкапсуляция также изолирует операции с типами данных, что позволяет:

- ограничить возможность возникновения ошибок;
- добавлять в реализации проверки на согласованность и другие средства отладки;
- сделать клиентский код яснее.

Инкапсулированный тип данных может быть использован любым клиентом, поэтому он расширяет язык Java. Пропагандируемый нами стиль программирования основан на идее разбиения больших программ на небольшие модули, которые можно разрабатывать и отлаживать независимо. Этот подход повышает гибкость программ, т.к. ограничивает и локализует эффекты внесения изменений, и обеспечивает повторное использование кода, поскольку позволяет подставлять новые реализации типа данных для повышения производительности, точности и экономии памяти. Этот же принцип применим и во многих других областях. Мы часто пользуемся инкапсуляцией при использовании системных библиотек. Новые версии системы Java часто содержат новые реализации различных типов данных или библиотек статических методов, но *API-интерфейсы не меняются*. В контексте изучения алгоритмов и структур данных существует сильное и постоянное стремление разрабатывать все лучшие алгоритмы, поскольку производительность *всех* клиентов можно повысить, подставив улучшенную реализацию АТД и не меняя код *ни одного* клиента. Ключ к успеху модульного программирования — поддержание *независимости* между модулями. Для этого мы настаиваем, чтобы API-интерфейс был *единственной* точкой сопряжения между клиентом и реализацией. *Нет необходимости знать, как реализован тип данных, чтобы использовать его*, а при реализации типа данных *можно считать, что клиент не знает ничего, кроме API-интерфейса*. Инкапсуляция является ключом к использованию обоих этих преимуществ.

Проектирование API-интерфейсов

Одним из наиболее важных и сложных шагов в построении современного ПО является проектирование API-интерфейсов. Для решения этой задачи нужны опыт, сосредоточенность и многократные повторения — но время, потраченное на проектирование хорошего API-интерфейса, обязательно окупится временем, сэкономленным при отладке и повторном использовании кода. При написании небольшой программки определение API-интерфейса может показаться излишним занятием, но создание *каждой* программы следует рассматривать с точки зрения повторного использования ее кода в будущем. В идеале API-интерфейс должен четко формулировать поведение типа для всех возможных входных данных, включая побочные эффекты, и еще нужно программное обеспечение для проверки, что реализации соответствуют спецификациям. К сожалению, в теории вычислительной техники получен фундаментальный результат, известный как *проблема спецификации*, который гласит, что эта цель в реальности *недостижима*. Коротко говоря, такая спецификация должна быть написана на формальном языке, аналогичного языкам программирования, а задача определения, выполняют ли две программы одно и то же вычисление, математически *неразрешима*. Поэтому наши API-интерфейсы содержат краткие описания на “человеческом” языке о множестве значений в абстрактном типе данных, а также список конструкторов и методов экземпляров — опять-таки с “человеческими” описаниями их назначения, включая и побочные эффекты. Для проверки API-интерфейса мы всегда включаем в сопровождающий их текст примеры клиентского кода. Наличие такого нестрогого определения чревато многочисленными опасностями при создании любого API-интерфейса.

- API-интерфейс может оказаться *слишком трудным для реализации*, а то и невозможным.
- API-интерфейс может оказаться *слишком трудным для использования* — вплоть до того, что клиентский код может стать сложнее, чем без API-интерфейса.
- API-интерфейс может оказаться *слишком узким*, т.е. не содержать методы, необходимые клиентам.
- API-интерфейс может оказаться *слишком широким*, т.е. включать множество методов, не нужных ни одному клиенту. Эта опасность, пожалуй, наиболее распространенная, и ее непросто избежать. Размер API-интерфейса со временем обычно растет, т.к. в существующий API-интерфейс нетрудно добавлять новые методы, но *трудно* удалять методы, не затрагивая существующие клиенты.
- API-интерфейс может оказаться *слишком общим*, т.е. не содержащим полезных абстракций.
- API-интерфейс может оказаться *слишком конкретным*, т.е. содержащим настолько подробные или раздробленные абстракции, что они оказываются бесполезными.
- API-интерфейс может оказаться *слишком зависимым от конкретного представления* и, таким образом, не освобождать код клиента от деталей использования этого представления. Этой опасности также трудно избежать, поскольку в основе разработки реализации, естественно, лежит представление.

Перечисленные соображения иногда кратко формулируются в виде своеобразного девиза: *предоставляйте клиентам только методы, которые им нужны, и ничего более.*

Алгоритмы и абстрактные типы данных

Абстракция данных естественно способствует изучению алгоритмов, т.к. она помогает сформировать среду, в которой можно точно задать, что алгоритм должен выполнять и как клиент должен использовать этот алгоритм. В данной книге алгоритм обычно представляет собой реализацию метода экземпляра в абстрактном типе данных. Например, наш пример с белым списком в начале этой главы естественным образом переводится в клиент АТД с помощью следующих операций:

- создание множества из массива заданных значений;
- определение, принадлежит ли этому множеству заданное значение.

Эти операции инкапсулированы в АТД `StaticSETofInts`, который приведен на рис. 1.2.26 вместе с типичным клиентом `Whitelist`. АТД `StaticSETofInts` — особый случай более общего и более полезного АТД *таблицы имен*, который будет рассмотрен в главе 3. Двоичный поиск — один из нескольких рассматриваемых нами алгоритмов, которые пригодны для реализации этих АТД. Сравнение с реализацией `BinarySearch` в листинге 1.1.4 показывает, что данная реализация приводит к более ясному и полезному клиентскому коду. Например, АТД `StaticSETofInts` обязательно сортирует массив перед вызовом `rank()`. Абстрактный тип данных отделяет клиент от реализации и облегчает *любому* клиенту задачу использования непростого алгоритма бинарного поиска: для этого надо просто следовать API-интерфейсу (клиенты метода `rank()` из класса `BinarySearch` должны знать о необходимости предварительной сортировки массива). Отбор по белому списку — один из многих клиентов, который использует бинарный поиск.

API-интерфейс

public class **StaticSETofInts**	
StaticSETofInts(int[] a)	*создание множества из значений массива* `a[]`
boolean contains(int key)	*принадлежит ли* `key` *множеству?*

Типичный клиент

```
public class Whitelist
{
   public static void main(String[] args)
   {
      int[] w = In.readInts(args[0]);
      StaticSETofInts set = new StaticSETofInts(w);
      while (!StdIn.isEmpty())
      {  // Чтение ключа и вывод, если его нет в белом списке.
         int key = StdIn.readInt();
         if (!set.contains(key))
            StdOut.println(key);
      }
   }
}
```

Реализация

```
import java.util.Arrays;
public class StaticSETofInts
{
  private int[] a;
  public StaticSETofInts(int[] keys)
  {
    a = new int[keys.length];
    for (int i = 0; i < keys.length; i++)
      a[i] = keys[i]; // копия на всякий случай
    Arrays.sort(a);
  }
  public boolean contains(int key)
  {  return rank(key) != -1;  }
  private int rank(int key)
  {  // Бинарный поиск.
    int lo  = 0;
    int hi = a.length - 1;
    while (lo <= hi)
    {  // Ключ находится в a[lo..hi]
       // или отсутствует.
      int mid = lo + (hi - lo) / 2;
      if      (key < a[mid]) hi = mid - 1;
      else if (key > a[mid]) lo = mid + 1;
      else                   return mid;
    }
    return -1;
  }
}
```

Применение

```
% java Whitelist
 largeW.txt < largeT.txt
499569
984875
295754
207807
140925
161828
 ...
```

Рис. 1.2.26. Бинарный поиск, оформленный в виде объектно-ориентированной программы (АТД для поиска во множестве целых чисел)

Любая Java-программа представляет собой набор статических методов и/или реализацию типа данных. В настоящей книге мы уделяем основное внимание реализациям *абстрактных* типов данных, таких как `StaticSETofInts`, где на первом плане находятся операции, а представление данных скрыто от клиента. Как видно из рассмотренного примера, абстракция данных позволяет:

- точно указать, что алгоритмы могут предоставить клиентам;
- отделить реализации алгоритмов от клиентского кода;
- разработать слои абстракции, где хорошо освоенные алгоритмы можно использовать для разработки других алгоритмов.

Эти свойства желательны для *любого* способа описания алгоритмов — хоть на естественном языке, хоть на псевдокоде. Заключая механизм Java-классов в рамки абстракции данных, мы почти ничего не теряем, зато многое приобретаем — работающий код, который мы можем проверить и использовать для сравнения производительности в различных клиентах.

Наследование интерфейса

В языке Java имеется поддержка определения взаимосвязей между объектами, которое называется *наследованием*. Эти механизмы широко применяются разработчиками программного обеспечения, и поэтому подробно изучаются в курсе проектирования ПО. Первым мы рассмотрим механизм наследования, который называется созданием *подтипов* и позволяет указывать взаимосвязь между различными классами. Для этого в *интерфейсе* описывается набор общих методов, который должен содержать каждый реализующий класс. Интерфейс — это просто список наследуемых методов. Например, вместо использования нашего неформального API-интерфейса интерфейс для типа `Date` можно сформулировать следующим образом:

```
public interface Datable
{
   int month();
   int day();
   int year();
}
```

а затем указывать этот интерфейс в коде реализации:

```
public class Date implements Datable
{
   // код реализации (тот же, что и раньше)
}
```

чтобы компилятор Java мог проверить соответствие этому интерфейсу. Добавление кода `implements Datable` в любой класс, реализующий методы `month()`, `day()` и `year()`, гарантирует любому клиенту, что объект данного класса может вызывать эти методы. Этот принцип называется *наследованием интерфейса*: реализующий класс *наследует* интерфейс. Наследование интерфейса позволяет писать клиентские программы, которые могут обрабатывать объекты *любого* типа, реализующие этот интерфейс (даже типа, который только планируется создать) — для этого просто вызываются методы, указанные в интерфейсе. Мы могли бы использовать наследование интерфейсов вместо наших менее формальных API-интерфейсов, но не будет этого делать, чтобы не привязываться к конкретным высокоуровневым языковым механизмам, которые не обязательны для пони-

мания алгоритмов, и чтобы не грузиться дополнительным багажом в виде интерфейсных файлов. Однако существуют ситуации, в которых соглашения Java делают привлекательным применение интерфейсов: мы будем использовать их для *сравнения* и *итерации*, как указано в табл. 1.2.3, и рассмотрим их подробнее, когда будем знакомиться с этими концепциями.

Таблица 1.2.3. Java-интерфейсы, используемые в данной книге

	Интерфейс	Методы	Раздел
Сравнение	`java.lang.Comparable`	`compareTo()`	2.1
	`java.util.Comparator`	`compare()`	2.5
Итерация	`java.lang.Iterable`	`iterator()`	1.3
	`ava.util.Iterator`	`hasNext()` `next()` `remove()`	1.3

Наследование реализации

Java поддерживает и другой механизм наследования — *подклассы* — мощную технику, которая позволяет изменять поведение и добавлять возможности без переписывания текста классов с нуля. В этой технике определяется новый класс (*подкласс* или *производный класс*), который наследует методы экземпляров *и* переменные экземпляров от другого класса (*суперкласс* или *базовый класс*).

Подкласс содержит больше методов, чем суперкласс. Более того, подкласс может *переопределять* методы суперкласса. Подклассы широко используются системными программистами для создания так называемых *расширяемых* библиотек: один программист (в том числе и вы) может добавлять методы в библиотеку, созданную другим программистом (или бригадой системных программистов), по сути, повторно используя код из (возможно большой) библиотеки. Этот подход, к примеру, широко применяется при разработке графических пользовательских интерфейсов, когда повторно используется значительный объем кода, необходимый для предоставления всех возможностей, необходимых пользователю (раскрывающиеся меню, перетаскивание мышью, доступ к файлам и т.д.).

Применение подклассов одобряется не всеми системными и прикладными программистами (преимущества наследования интерфейсов сомнительны), и мы будем избегать его в настоящей книге, т.к. оно обычно мешает инкапсуляции. Рудименты этого подхода встроены в Java, и поэтому полностью избежать их не удастся: например, все классы в Java порождены от класса `Object`. Данная структура позволяет ввести "соглашение", что каждый класс должен содержать реализацию `getClass()`, `toString()`, `equals()`, `hashCode()` и ряда других методов, которые не будут использоваться в данной книге. Вообще говоря, каждый класс *наследует* эти методы от класса `Object` через цепочку подклассов, поэтому любой клиент может использовать их для любого объекта. Мы обычно переопределяем в своих классах методы `toString()`, `equals()` и `hashCode()`, поскольку стандартная реализация в классе `Object` обычно делает не то, что нам нужно (табл. 1.2.4). Сейчас мы рассмотрим методы `toString()` и `equals()`, а с методом `hashCode()` познакомимся в разделе 3.4.

Таблица 1.2.4. Методы, унаследованные от класса `Object`, которые используются в данной книге

Метод	Назначение	Раздел
`Class getClass()`	К какому классу принадлежит объект?	1.2
`String toString()`	Строковое представление объекта.	1.1
`boolean equals(Object that)`	Совпадает ли данный объект с `that`?	1.2
`int hashCode()`	Хеш-код объекта	3.4

Преобразование в строку

По соглашению все Java-типы наследуют от класса `Object` метод `toString()`, и поэтому любой клиент может вызвать метод `toString()` для любого объекта. На этом соглашении основано автоматическое преобразование в Java одного операнда операции + в тип `String`, если другой операнд имеет тип `String`. Если у типа данных объекта нет реализации метода `toString()`, то вызывается стандартная реализация из типа `Object` — но она, как правило, бесполезна, т.к. возвращает строковое представление адреса объекта в памяти. Поэтому мы обычно включаем реализации метода `toString()`, которые переопределяют стандартную реализацию во всех разрабатываемых нами классах, как показано для типа `Date` в листинге 1.2.5. Обычно реализации `toString()` просты и неявно (с помощью операции +) вызывают метод `toString()` для всех переменных экземпляров.

Типы-оболочки

В Java имеются встроенные ссылочные типы, которые называются *типами-оболочками* — по одному для каждого примитивного типа: `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long` и `Short`, которые соответствуют типам `boolean`, `byte`, `char`, `double`, `float`, `int`, `long` и `short`. Эти классы содержат в основном статические методы наподобие `parseInt()`, но, кроме того, наследуют методы экземпляров `toString()`, `compareTo()`, `equals()` и `hashCode()`. Java выполняет автоматическое преобразование примитивных типов в типы-оболочки, когда это нужно (см. подраздел “Автоупаковка” в разделе 1.3). Например, при попытке конкатенации значения `int` со значением `String` оно преобразуется в тип `Integer`, для которого имеется метод `toString()`.

Равенство

Что означает утверждение о том, что два объекта равны? При проверке равенства с помощью операции `a == b`, если `a` и `b` — ссылочные переменные одного типа, выполняется проверка на эквивалентность идентичностей, т.е. на равенство *ссылок*. Однако обычно клиентам требуется знать, совпадают ли *значения типа данных* (состояние объекта), или даже реализовать особое правило для конкретного типа. Отправной точкой могут быть существующие в Java реализации как для стандартных типов вроде `Integer`, `Double` и `String`, так и для более сложных типов наподобие `File` и `URL`. При работе с этими типами данных можно просто использовать встроенные реализации. Например, если `x` и `y` — значения типа `String`, то `x.equals(y)` истинно тогда и только тогда, когда `x` и `y` имеют одинаковую длину и совпадают в каждой символьной позиции. При определении собственных типов данных, подобных `Date` или `Transaction`, необходимо переопределить метод `equals()`.

В Java действует соглашение, что метод `equals()` должен выражать *отношение эквивалентности*, т.е. иметь следующие свойства:

- *рефлексивность* — `x.equals(x)` истинно для любого `x`;
- *симметричность* — `x.equals(y)` истинно тогда и только тогда, когда истинно `y.equals(x)`;
- *транзитивность* — если истинны выражения `x.equals(y)` и `y.equals(z)`, то истинно и `x.equals(z)`.

Кроме того, это отношение должно принимать в качестве аргумента тип `Object` и удовлетворять следующим требованиям:

- *согласованность* — несколько вызовов `x.equals(y)` должны выдать одно и то же значение, если ни один из объектов не изменяется;
- *отличие от* `null` — `x.equals(null)` всегда возвращает `false`.

Эти требования вполне естественны, однако их обеспечение с учетом соглашений Java, да еще и без излишних трудов при реализации, может быть непростой задачей, как видно из листинга 1.2.6. Для этого выполняется описанный ниже многошаговый процесс.

- Если ссылка на данный объект совпадает со ссылкой на объект аргумента, возвращается `true`. Эта проверка сокращает объем работы, необходимой для выполнения дальнейших проверок.
- Если аргумент является пустой ссылкой, возвращается `false`, чтобы удовлетворить требованию оличия от `null` (и избежать перехода по нулевой ссылке).
- Если объекты не принадлежат одному и тому же классу, возвращается `false`. Для определения класса объекта используется метод `getClass()` и операция `==`, которая возвращает одинаковые ссылки для всех объектов одного класса.
- Аргумент приводится из типа `Object` к типу `Date` (это приведение возможно после предыдущей проверки).
- Если хотя бы одна переменная экземпляра не совпадает, возвращается `false`. Для других классов может применяться какое-то другое определение эквивалентности. Например, два объекта `Counter` эквивалентны, если равны их переменные экземпляров `count`.

Листинг 1.2.6. Переопределение методов `toString()` и `equals()` в определении типа данных

```
public class  Date
{
  private final int month;
  private final int day;
  private final int year;

  public Date(int m, int d, int y)
  {  month = m; day = d; year = y; }

  public int month()
  {  return month;  }
```

```
  public int day()
  {  return day;  }
  public int year()
  {  return year;  }
  public String toString()
  {  return month() + "/" + day() + "/" + year();  }
  public boolean equals(Object x)
  {
     if (this == x) return true;
     if (x == null) return false;
     if (this.getClass() != x.getClass()) return false;
     Date that = (Date) x;
     if (this.day != that.day)          return false;
     if (this.month != that.month)      return false;
     if (this.year != that.year)        return false;
     return true;
  }
}
```

Эту реализацию можно использовать в качестве модели при реализации метода equals() для любого другого типа. После одной такой реализации будет уже нетрудно создать equals() и для других типов.

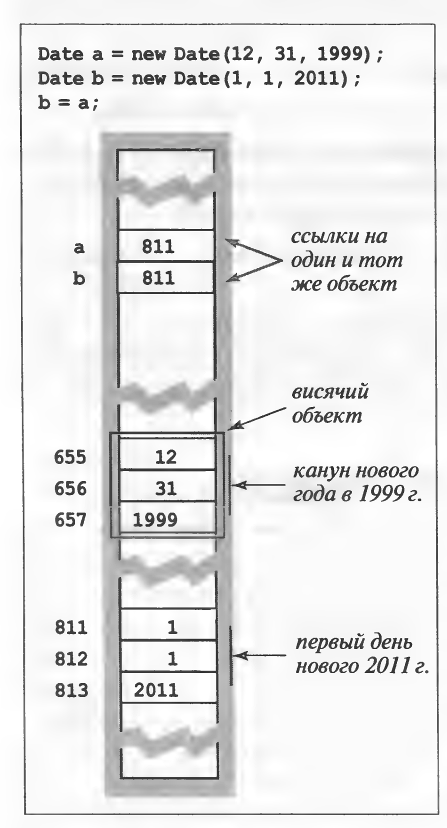


Рис. 1.2.27. Висячий объект

Управление памятью

Возможность присваивать новое значение ссылочной переменной приводит к возможности появления объектов, на которые не указывают никакие ссылки. Рассмотрим, к примеру, три оператора присваивания на рис. 1.2.27. После выполнения третьего оператора не только a и b указывают на один и тот же объект Date (01/01/2011), но и уже не существует ссылка на объект Date, которая была создана и использовалась для инициализации b. Единственная ссылка на данный объект находилась в переменной b, но она была перезаписана третьим оператором, после чего уже нет возможности снова обратиться к этому объекту. Такие объекты называются *покинутыми* или *висячими* (orphaned). Объекты становятся висячими и тогда, когда выходят из области видимости. Java-программы часто создают очень много объектов (и переменных, которые хранят значения примитивных типов), но в любой момент времени им нужно лишь небольшое их количество. Поэтому языкам и системам программирования необходимы механизмы *выделения* памяти для значений типов данных на время, пока они нужны, и *освобождения* памяти, когда они уже не требуются (после того как объект стал висячим). Управление памятью проще выполнять для при-

митивных типов, поскольку вся необходимая для этого информация известна еще на этапе компиляции. Java (как и большинство других систем) берет на себя заботу о резервировании памяти для переменных при их объявлении и освобождении этой памяти, когда они выходят из области видимости. Управление памятью для объектов сложнее: система может выделить память объекту в момент его создания, но она не знает точно, когда следует освободить память, связанную с каждым объектом, т.к. момент, когда объект становится ненужным, зависит от динамики программы.

Во многих языках (таких как C и C++) программист полностью отвечает как за выделение, так и за освобождение памяти. Это утомительное занятие, которое является источником многочисленных ошибок. Одной из самых заметных отличительных черт Java является возможность *автоматического* управления памятью. Чтобы освободить программистов от ответственности за управление памятью, система отслеживает висячие объекты и возвращает занятую ими память в пул свободной памяти. Такой возврат памяти называется *сборкой мусора*. В Java принята политика, что ссылки нельзя изменять. Эта политика позволяет Java эффективно выполнять сборку мусора. Программисты так пока и не пришли к общему мнению, окупается ли автоматическая сборка мусора отсутствием забот об управлении памятью.

Неизменяемость

Неизменяемый тип данных, такой как `Date`, характеризуется тем, что значение объекта никогда не меняется после его создания. А *изменяемый* тип данных, подобный `Counter` или `Accumulator`, содержит значения, предназначенные для изменения (табл. 1.2.5). В Java имеется механизм поддержки неизменяемости — модификатор `final`. При объявлении переменной с таким модификатором программист обещает присвоить значение этой переменной только один раз — либо в инициализаторе, либо в конструкторе. Код, который может изменить значение переменной `final`, приводит к появлению ошибки времени компиляции. В нашем коде мы используем модификатор `final` с переменными экземпляров, значения которых никогда не меняются. Эта политика служит для документирования того, что значение точно не изменится, предохраняет от случайного изменения и облегчает отладку программ. Например, значение `final` можно не выводить в трассировке, т.к. точно известно, что ее значение не меняется.

Таблица 1.2.5. Примеры изменяемых и неизменяемых типов

Изменяемый	Неизменяемый
`Counter`	`Date`
Массивы в Java	`String`

Тип данных наподобие `Date`, все переменные экземпляров в котором примитивны и описаны как `final`, является неизменяемым (по соглашению в коде, в котором не применяется наследование реализаций). Сделать ли тип данных неизменяемым — важное проектное решение, которое зависит от конкретного приложения. Для типов данных вроде `Date` цель абстракции состоит в инкапсуляции значений, которые не будут изменяться, и которые поэтому можно использовать в операторах присваивания и в качестве возвращаемых значений так же, как и примитивные типы (не беспокоясь об изменении значений). Программист, реализующий клиент типа `Date`, вполне может применять оператор `d = d0` с участием двух переменных `Date` — как и для переменных `double` или `int`. Но если `Date` — изменяемый тип, и значение `d0` должно измениться *после* при-

сваивания d = d0, то изменится *также* и значение d0: ведь обе ссылки указывают на один и тот же объект. А для таких типов данных, как Counter или Accumulator, эта абстракция как раз и предназначена для инкапсуляции изменяемых значений.

Вы уже встречались с этим различием в роли программиста клиентской программы — при использовании Java-массивов (изменяемые) и типа данных String (неизменяемый). При передаче объекта String в метод не надо было беспокоиться, что этот метод изменит в строке последовательность символов, но при передаче массива метод может свободно изменять содержимое массива. Объекты String неизменяемы потому, что обычно их *не нужно* изменять, а массивы изменяемы потому, что обычно изменять их значения *необходимо*. Но бывают ситуации, когда необходимы изменяемые строки (предназначение класса StringBuilder) и когда необходимы неизменяемые массивы (предназначение класса Vector, который вскоре будет рассмотрен). В общем случае с неизменяемыми типами легче работать и сложнее что-то напутать, чем с изменяемыми, т.к. область кода, где могут изменяться их значения, значительно *уже*. Код, содержащий неизменяемые типы, существенно легче отлаживать, поскольку легче гарантировать, что переменные в клиентском коде, который их использует, находятся в согласованном между собой состоянии. А в случае применения изменяемых типов постоянно приходится отвечать на вопрос, где и когда меняются их значения.

Недостаток неизменяемости в том, что *для каждого нового значения приходится создавать новый объект*. Обычно в этом нет ничего страшного, поскольку сборщики мусора Java спроектированы с учетом такой ситуации. Другой недостаток неизменяемости вытекает из того факта, что, к сожалению, модификатор final гарантирует неизменяемость, только если переменные экземпляров являются примитивными типами, а не ссылочными. При наличии модификатора final у переменной экземпляра ссылочного типа ее значение (ссылка на объект) действительно не изменится, поскольку будет указывать на один и тот же объект, но значение самого объекта *может* измениться. Например, вот этот код *не* реализует неизменяемый тип:

```
public class Vector
{
   private final double[] coords;

   public Vector(double[] a)
   {  coords = a; }
   ...
}
```

Клиентская программа может создать объект Vector, указав элементы массива, а затем (в обход API-интерфейса) изменить элементы этого массива после его создания:

```
double[] a = { 3.0, 4.0 };
Vector vector = new Vector(a);
a[0] = 0.0;  // В обход общедоступного API.
```

Переменная экземпляра coords[] объявлена как private и final, но тип Vector является изменяемым, т.к. клиенту доступна ссылка на данные. Неизменяемость следует учитывать при любом проектировании типов данных и указывать в API-интерфейсе, чтобы программисты клиентов знали, что значения объектов не меняются. В данной книге мы будем применять неизменяемость для подтверждения правильности наших алгоритмов. Например, если бы тип данных, используемый для алгоритма бинарного поиска, был изменяемым, то клиенты могли бы нарушить предположение, что массив, в котором выполняется поиск, упорядочен.

Контрактное проектирование

В завершение мы кратко рассмотрим механизмы языка Java, которые позволяют проверять предположения о программе *во время ее выполнения*. Для этой цели мы будем применять два следующих механизма:

- исключения, которые обычно обрабатывают непредвиденные ситуации *вне* управления выполнением;
- утверждения, которые проверяют сформулированные предположения *внутри* разрабатываемого кода.

Разумное использование и исключений, и утверждений — хорошая тренировка в программировании. Мы применяем их в книге нечасто — для экономии места, но в коде на сайте книги они встречаются гораздо чаще. Этот код соответствует значительному объему комментариев в тексте, которые посвящены исключительным условиям и утверждаемым инвариантам в каждом алгоритме.

Исключения и ошибки

Исключения и *ошибки* — это разрушительные события, которые возникают во время выполнения программы, часто с выводом сообщений об ошибках. Выполняемое при этом действие называется *генерацией исключения* или *генерацией ошибки*. Мы уже познакомились с исключениями, генерируемыми системными методами, при изучении базовых возможностей Java; типичные примеры — `StackOverflowError`, `ArithmeticException`, `ArrayIndexOutOfBoundsException`, `OutOfMemoryError` и `NullPointerException`. Можно создавать и собственные исключения. Простейшее из них — `RuntimeException`, которое прекращает выполнение программы и выводит сообщение об ошибке:

```
throw new RuntimeException("Сообщение об ошибке.");
```

Широко распространен подход *быстрого сбоя* (fail fast), основанный на предположении, что ошибку легче обнаружить, если исключение генерируется как можно скорее после обнаружения ошибки (в отличие от игнорирования ошибки и генерации ошибки как-нибудь потом).

Утверждения

Утверждение представляет собой логическое выражение, которое должно быть истинно в конкретной точке программы. Если утверждение неверно, программа завершается с выдачей сообщения об ошибке. Мы используем утверждения для уверенности в правильности программ и для документирования. Пусть, например, вычислено некоторое значение, которое можно использовать в качестве индекса в массиве. Если это значение отрицательно, то однажды оно станет причиной исключения `ArrayIndexOutOfBoundsException`. Но, поместив в код оператор `assert index >= 0;`, можно точнее определить место возникновения ошибки. Кроме того, можно добавить необязательное информативное сообщение:

```
assert index >= 0 : "Отрицательный индекс в методе X";
```

которое поможет выявить ошибку. По умолчанию утверждения отключены. Включить их можно в командной строке с помощью флага `-enableassertions` (сокращенно `-ea`). Утверждения предназначены для отладки; программа не должна использовать их

в рабочем режиме, поскольку они могут быть отключены. В курсе системного программирования вы научитесь применять утверждения для того, чтобы код *не* завершался по системной ошибке или не попал в бесконечный цикл. Одна из моделей, называемая моделью *контрактного проектирования* (design by contract) в программировании, как раз выражает эту идею. Тот, кто проектирует тип данных, формулирует *входное условие* (условие, которое клиент обещает выполнять при вызове метода), *выходное условие* (условие, которое реализация обещает выполнять при возврате из метода) и *побочные эффекты* (любые другие изменения состояния, причиной которых может стать метод). Во время разработки все эти условия можно проверять с помощью утверждений.

Резюме

Языковые механизмы, рассмотренные в настоящем разделе, демонстрируют, что эффективное проектирование типов данных приводит к нетривиальным вопросам, ответы на которые получить не всегда легко. Эксперты все еще спорят об оптимальных способах поддержки некоторых рассмотренных здесь проектных принципов. Почему в Java не разрешено передавать функции в качестве аргументов? Почему Matlab копирует массивы, передаваемые в качестве аргументов функциям? Как было сказано еще в главе 1, жалобы на какие-то возможности языка программирования ведут прямиком на скользкий путь к созданию нового языка. Если вы не собираетесь идти по ней, лучше воспользоваться повсеместно доступными языками.

В большинстве систем имеются обширные библиотеки, которыми, несомненно, следует пользоваться при каждой возможности, но зачастую можно упростить и защитить клиентский код, создавая абстракции, которые легко переносятся в другие языки. Ваша главная цель — разработка таких типов данных, чтобы основная работа выполнялась на уровне абстракции, соответствующей решаемой задаче.

В табл. 1.2.6 приведена сводка различных рассмотренных нами видов Java-классов.

Таблица 1.2.6. Java-классы (реализации в виде типов данных)

Вид класса	Примеры	Характеристики
Статические методы	`Math StdIn StdOut`	Без переменных экземпляров
Неизменяемый абстрактный тип данных	`Date Transaction String Integer`	Все переменные экземпляров `private`
		Все переменные экземпляров `final`
		Безопасное копирование для ссылочных типов
		Примечание: эти характеристики необходимы, но недостаточны
Изменяемый абстрактный тип данных	`Counter Accumulator`	Все переменные экземпляров `private`
		Не все переменные экземпляров `final`
Абстрактный тип данных с побочными эффектами ввода-вывода	`VisualAccumulator In Out Draw`	Все переменные экземпляров `private`
		Методы экземпляров выполняют ввод-вывод

Вопросы и ответы

Вопрос. **Зачем вообще нужна абстракция данных?**

Ответ. Она помогает создавать надежный и корректный код.

Вопрос. **Зачем делать различие между примитивными и ссылочными типами? Почему не оставить только ссылочные типы?**

Ответ. Производительность. Для тех алгоритмистов, которым нравится игнорировать это различие, в Java имеются ссылочные типы `Integer`, `Double` и т.д., которые соответствуют примитивным типам. Примитивные типы ближе к типам данных, которые поддерживаются оборудованием компьютера, поэтому программы, которые используют их, обычно работают быстрее, чем программы, в которых применяются соответствующие ссылочные типы.

Вопрос. **Типы данных *обязательно* должны быть абстрактными?**

Ответ. Нет. В Java имеются еще описатели `public` и `protected`, которые позволяют некоторым клиентам обращаться непосредственно к переменным экземпляров. Как сказано в тексте, преимущества разрешения клиентскому коду напрямую обращаться к данным нивелируются недостатками зависимости от конкретного представления, поэтому в нашем коде все переменные экземпляров описаны как `private`. Кроме того, мы иногда применяем приватные методы экземпляров, чтобы получить возможность использовать код несколькими общедоступными методами.

Вопрос. **Что случится, если не написать операцию `new` при создании объекта?**

Ответ. В Java это выглядит так, как будто вы хотите вызвать статический метод, который возвращает значение типа объекта. Поскольку такой метод не определен, возникнет сообщение об ошибке, как и в случае обращения к не определенному имени. При компиляции оператора

```
Counter c = Counter("test");
```

вы получите следующее сообщение об ошибке:

```
cannot find symbol                     не найден символ
symbol  : method Counter(String)       символ  : метод Counter(String)
```

Такое же сообщение об ошибке генерируется и при вызове конструктора с неверным количеством аргументов.

Вопрос. **Что случится, если не написать операцию `new` при создании массива объектов?**

Ответ. Операция `new` необходима для каждого создаваемого объекта, поэтому при создании массива из N объектов понадобится $N+1$ операций `new`: одна для всего массива и по одной для каждого объекта. Если вы забудете создать массив:

```
Counter[] a;
a[0] = new Counter("проверка");
```

то получите то же сообщение, что и при попытке присваивания значения любой неинициализированной переменной:

```
variable a might not have been initialized
переменная может быть не инициализирована
     a[0] = new Counter("проверка");
     ^
```

Но если вы забудете записать операцию `new` при создании объекта в массиве, а затем попытаетесь использовать его для вызова метода:

```
Counter[] a = new Counter[2];
a[0].increment();
```

возникнет исключение `NullPointerException`.

Вопрос. **Почему бы не печать объекты с помощью конструкции `StdOut.println(x.toString())`?**

Ответ. Этот код работает без проблем, но Java снимает с нас ответственность за его написание: для любого объекта автоматически вызывается метод `toString()`, т.к. `println()` вызывает метод, который принимает аргумент типа `Object`.

Вопрос. **Что такое *указатель*?**

Ответ. Хороший вопрос. Как и Java-ссылки, *указатель* можно рассматривать как машинный адрес. Во многих языках программирования указатель является примитивным типом данных, который программист может обрабатывать многими способами. Однако работа с указателями по своей сути может привести ко многим ошибкам, поэтому необходимо тщательно спроектировать операции над указателями, чтобы помочь программистам избежать этих ошибок. Java развивает эту точку зрения до абсолюта (как поступают и разработчики многих других современных языков программирования). В Java имеется лишь *один* способ создать ссылку (`new`) и лишь *один* способ изменить ссылку (оператор присваивания). То есть программист может только создавать ссылки и копировать их. На программистском жаргоне Java-ссылки называются *безопасными указателями*, т.к. Java может гарантировать, что каждая ссылка указывает на объект заданного типа (и может определить, какие объекты не используются, для сборки мусора). Программисты, привыкшие писать код, который непосредственно манипулирует указателями, считают, что в Java просто нет указателей, но многие все еще спорят, нужно ли иметь небезопасные указатели.

Вопрос. **Где можно найти дополнительную информацию о том, как в Java реализованы ссылки и сборка мусора?**

Ответ. Одна система Java может полностью отличаться от другой. Например, ссылки часто реализуются как указатели (машинный адрес) или как *дескрипторы* (указатель на указатель). Первый вариант обеспечивает более быстрый доступ к данным, а второй удобнее для сборки мусора.

Вопрос. **Что в точности означает импорт имени?**

Ответ. Не особенно много: это просто сокращает объем кода. Можно не использовать оператор `import` и везде в коде вместо `Arrays` писать `java.util.Arrays`.

Вопрос. **Что такое проблема наследования реализации?**

Ответ. Создание подтипов усложняет модульное программирование по двум причинам. Во-первых, любое изменение в суперклассе влияет на все подклассы. Подкласс невозможно разработать *независимо* от суперкласса — вообще-то он *полностью зависим* от своего суперкласса. Эта проблема называется проблемой *уязвимости базового класса*. Во-вторых, код подкласса, имея доступ к переменным экземпляров, может исказить назначение кода суперкласса. Например, проектировщик такого класса, как `Counter`, для системы голосования может принять все возможные меры для того, чтобы счетчик этого класса увеличивался только на

единицу. Но подкласс имеет полный доступ к переменной счетчика и может занести в нее произвольное значение.

Вопрос. **Как сделать класс неизменяемым?**

Ответ. Чтобы гарантировать неизменяемость типа данных, который содержит переменную экземпляра изменяемого вида, необходимо делать локальную копию, которая называется *безопасной копией*. И даже этого может оказаться недостаточно. Создание копии — это одна трудность, а защита от изменения значений одним из методов экземпляров — еще одна.

Вопрос. **Что такое `null`?**

Ответ. Это литеральное значение, которое не указывает ни на какой объект. Вызов метода с помощью нулевой ссылки не имеет смысла и приводит к возникновению исключения `NullPointerException`. Если вы получите сообщение о такой ошибке, проверьте, правильно ли инициализирует конструктор все переменные экземпляров.

Вопрос. **Может ли класс, реализующий тип данных, содержать статический метод?**

Ответ. Конечно. Например, все классы содержат метод `main()`. Кроме того, вполне естественно применять статические методы для операций с несколькими объектами, когда ни один из них нельзя естественно выделить для вызова метода. Например, в классе `Point` можно определить примерно такой статический метод:

```
public static double distance(Point a, Point b) // Расстояние между точками
{
   return a.distTo(b);
}
```

Часто добавление таких методов может повысить ясность клиентского кода.

Вопрос. **Есть ли еще какие-то виды переменных, кроме параметров, локальных переменных и переменных экземпляров?**

Ответ. Если добавить в объявление класса ключевое слово `static` (вне любого типа), то появится совершенно не похожий на описанные здесь вид переменной — *статическая переменная*. Как и переменные экземпляров, статические переменные доступны любому метода класса, но они не связаны ни с каким объектом. В старых языках программирования такие переменные назывались *глобальными переменными*, из-за глобальности их области видимости. В современных языках программирования больше внимания уделяется ограничению области видимости, и поэтому такие переменные применяются реже. При их использовании нужна повышенная осторожность.

Вопрос. **Что такое *не рекомендуемый* метод?**

Ответ. Это метод, который уже не поддерживается полностью, но остается в API-интерфейсе для поддержки совместимости. Например, когда-то в Java появился метод `Character.isSpace()` (проверка, является ли символ пробельным), и программисты писали программы, которые полагались на поведение этого метода. Когда разработчики языка Java позже захотели добавить поддержку дополнительных пробельных символов из Unicode, они не могли изменить поведение `isSpace()`, не нарушив работу клиентских программ, и поэтому добавили другой метод `Character.isWhiteSpace()`, а старый метод объявили не рекомендуемым. Конечно, с течением времени такая практика усложня-

ет API-интерфейсы. Иногда не рекомендуемыми объявляются целые классы. Например, в Java не рекомендуется к применению класс `java.util.Date`, чтобы улучшить поддержку интернационализации.

Упражнения

1.2.1. Напишите клиент класса `Point2D`, который принимает в командной строке целое значение `N`, генерирует `N` случайных точек на единичном квадрате и вычисляет расстояние, разделяющее *самую близкую* пару точек.

1.2.2. Напишите клиент класса `Interval1D`, который принимает в командной строке целое значение `N`, читает из стандартного ввода `N` интервалов (каждый из которых определен парой значений `double`) и выводит все пересекающиеся пары.

1.2.3. Напишите клиент класса `Interval2D`, который принимает в командной строке аргументы `N`, `min` и `max` и генерирует на единичном квадрате `N` случайных двумерных интервалов, ширины и высоты которых равномерно распределены между `min` и `max`. Начертите их на `StdDraw` и выведите количество пересекающихся пар интервалов и количество интервалов, содержащихся один в другом.

1.2.4. Что выведет следующий фрагмент кода?

```
String string1 = "hello";
String string2 = string1;
string1 = "world";
StdOut.println(string1);
StdOut.println(string2);
```

1.2.5. Что выведет следующий фрагмент кода?

```
String s = "Hello World";
s.toUpperCase();
s.substring(6, 11);
StdOut.println(s);
```

Ответ: `"Hello World"`. Объекты `String` неизменяемы: строковые методы возвращают новый объект `String` с соответствующим значением и не изменяют значение вызвавшего их объекта. Этот код игнорирует возвращаемые объекты и просто выводит исходную строку. Для вывода `"WORLD"` нужны операторы `s = s.toUpperCase()` и `s = s.substring(6, 11)`.

1.2.6. Строка `s` называется ***циклическим вращением*** строки `t`, если она совпадает с `t` после циклического сдвига символов на любое количество позиций. Например, строка `ACTGACG` является циклическим сдвигом строки `TGACGAC`, и наоборот. Выявление таких ситуаций важно в изучении генных последовательностей. Напишите программу, которая проверят, являются ли заданные строки `s` и `t` циклическими вращениями одна другой. *Совет*: решение можно записать в одну строку с помощью методов `indexOf()`, `length()` и конкатенации.

1.2.7. Что возвращает следующая рекурсивная функция?

```
public static String mystery(String s)
{
   int N = s.length();
   if (N <= 1) return s;
   String a = s.substring(0, N/2);
   String b = s.substring(N/2, N);
   return mystery(b) + mystery(a);
}
```

1.2.8. Пусть `a[]` и `b[]` — целочисленные массивы, содержащие миллионы чисел. Что делает следующий код? Достаточно ли он эффективен?

```
int[] t = a; a = b; b = t;
```

Ответ: он обменивает массивы. Вряд ли можно написать более эффективный код, т.к. здесь копируются только ссылки, без необходимости копирования миллионов элементов.

1.2.9. Добавьте в класс `BinarySearch` (листинг 1.1.4) тип `Counter` для подсчета общего количества ключей, просмотренных во время всех поисков, с последующим выводом этого количества после завершения всех поисков. *Совет*: создайте переменную `Counter` в методе `main()` и передайте ее в качестве аргумента методу `rank()`.

1.2.10. Разработайте класс `VisualCounter`, в котором имеются операции как увеличения, так и уменьшения счетчика на единицу. Конструктор должен принимать два аргумента — `N` и `max`, где `N` задает максимальное количество операций, а `max` — максимальное абсолютное значение для счетчика. В качестве побочного эффекта класс должен создавать чертеж со значением счетчика при каждом его изменении.

1.2.11. Разработайте реализацию типа `SmartDate` из нашего API-интерфейса `Date`, которая должна генерировать исключение, если ей попадается несуществующая дата.

1.2.12. Добавьте в тип данных `SmartDate` метод `dayOfTheWeek()`, возвращающий для даты строковое значение `понедельник`, `вторник`, `среда`, `четверг`, `пятница`, `суббота` или `воскресенье`. Можно считать, что даты принадлежат XXI веку.

1.2.13. Разработайте реализацию типа данных `Transaction` по образу реализации типа `Date` (рис. 1.2.22).

1.2.14. Разработайте реализацию метода `equals()` для типа `Transaction` по образу метода `equals()` для типа `Date` (листинг 1.2.5).

Творческие задачи

1.2.15. *Файловый ввод*. Разработайте возможную реализацию статического метода `readInts()` из класса `In` (который мы будем использовать для различных клиентов тестирования — например, бинарного поиска), основанную на методе `split()` из класса `String`.

Решение:

```
public static int[] readInts(String name)
{
   In in = new In(name);
   String input = StdIn.readAll();
    String[] words = input.split("\\s+");
   int[] ints = new int[words.length;
   for int i = 0; i < word.length; i++)
      ints[i] = Integer.parseInt(words[i]);
   return ints;
}
```

Другая реализация будет рассмотрена в разделе 1.3.

1.2.16. *Рациональные числа.* Реализуйте неизменяемый тип данных `Rational` для рациональных чисел, который поддерживает операции сложения, вычитания, умножения и деления.

public class **Rational**

	`Rational(int numerator, int denominator))` *создание из числителя и знаменателя*	
`Rational`	`plus(Rational b)`	*сумма данного числа и* `b`
`Rational`	`minus(Rational b)`	*разность данного числа и* `b`
`Rational`	`times(Rational b)`	*произведение данного числа и* `b`
`Rational`	`divides(Rational b)`	*частное данного числа и* `b`
`boolean`	`equals(Rational that)`	*равно ли данное число* `that`*?*
`String`	`toString()`	*строковое представление*

Чтобы не заботиться о переполнении (см. упражнение 1.2.17), используйте в качестве переменных экземпляров два значения `long`, которые представляют числитель и знаменатель. Для сокращения числителя и знаменателя на общий делитель используйте алгоритм Евклида (см. рис. 1.0.1). Добавьте клиент тестирования, который демонстрирует работоспособность всех написанных вами методов.

1.2.17. *Устойчивая реализация рациональных чисел.* Используйте утверждения для разработки реализации типа данных `Rational` (см. упражнение 1.2.16), которая будет защищена от переполнения.

1.2.18. *Накопитель с дисперсией.* Проверьте правильность следующего кода, который добавляет в класс `Accumulator` методы `var()` и `stddev()` и вычисляет среднее значение и дисперсию чисел, представленных как аргументы метода `addDataValue()`:

```
public class Accumulator
{
   private double m;
   private double s;
   private int N;

   public void addDataValue(double x)
   {
      N++;
      s = s + 1.0 * (N-1) / N * (x - m) * (x - m);
      m = m + (x - m) / N;
   }

   public double mean()
   {  return m;  }

   public double var()
   {  return s/(N - 1);  }

   public double stddev()
   {  return Math.sqrt(this.var());  }
}
```

Эта реализация не так чувствительна к ошибкам округления, как примитивная реализация, где сохраняется сумма квадратов чисел.

1.2.19. *Разбор строк*. Разработайте конструкторы с разбором строк для реализаций Date и Transaction из упражнения 1.2.13, которые принимают единый аргумент типа String с инициализирующими значениями в форматах, представленных в табл. 1.2.7.

Таблица 1.2.7. Форматы для разбора строк

Тип	Формат	Пример
Date	Целые числа, разделенные косыми чертами	5/22/1939
Transaction	Клиент, дата и сумма, разделенные пробельными символами	Turing 5/22/1939 11.99

Частичное решение:

```
public Date(String date)
{
   String[] fields = date.split("/");
   month = Integer.parseInt(fields[0]);
   day   = Integer.parseInt(fields[1]);
   year  = Integer.parseInt(fields[2]);
}
```

1.3. Контейнеры, очереди и стеки

Некоторые фундаментальные типы данных содержат *коллекции* объектов. А именно: множество значений типа представляет собой коллекцию объектов, а операции основаны на добавлении, удалении или просмотре объектов из этой коллекции. В настоящем разделе мы рассмотрим три таких типа: *контейнер*, *очередь* и *стек*. Они различаются правилом, определяющим объект, который удаляется или просматривается первым.

Контейнеры, очереди и стеки — фундаментальные типы и поэтому они широко используются. Мы будем применять их во многих реализациях данной книги. Кроме этой непосредственной пользы, код клиентов и реализаций из данного раздела служит введением в общий подход к разработке структур данных и алгоритмов.

Одна из целей этого раздела — подчеркнуть мысль, что способ представления объектов в коллекции непосредственно влияет на эффективность различных операций. Мы разработаем структуры данных для представления коллекции объектов, которые могут поддерживать эффективные реализации некоторых операций.

Вторая цель связана с вводом понятий *обобщений* и *итерации* — базовых концепций Java, которые существенно упрощают код клиентов. Это дополнительные механизмы языка программирования, которые не обязательны для понимания алгоритмов, но позволяют разрабатывать более ясный, компактный и элегантный клиентский код (и реализации алгоритмов), чем это было бы без них.

Третья цель настоящего раздела — знакомство со *связными* структурами данных и демонстрация их важности. В частности, классическая структура данных — *связный список* — позволяет реализовать контейнеры, очереди и стеки с эффективностью, не достижимой другим способами. Изучение строения и работы связных списков — первый важный шаг в исследовании алгоритмов и структур данных.

Для каждого из перечисленных трех типов мы рассмотрим API-интерфейсы и примеры клиентских программ, а затем возможные представления значений типа данных и реализации операций над этими типами данных. Этот сценарий еще не раз повторится в данной книге (с более сложными структурами данных). Приведенные здесь реализации служат образцами для реализаций, приведенных далее в данной книге, но и сами по себе они достойны тщательного изучения.

API-интерфейсы

Как обычно, мы начнем обсуждение абстрактных типов данных для коллекций с определения их API-интерфейсов, которые приведены на рис. 1.3.1. Каждый из них содержит конструктор без аргументов, метод добавления элемента в коллекцию, метод для проверки, пуста ли коллекция, и метод, возвращающий размер коллекции. У типов `Stack` и `Queue` имеются методы для удаления из коллекции конкретного элемента. Кроме этих основных типов, в рассматриваемых API-интерфейсах отражены две возможности Java, которые будут рассмотрены на следующих нескольких страницах — *обобщенные* и *итерируемые коллекции*.

Контейнер

public class **Bag<Item>** implements Iterable<Item>		
	`Bag()`	*создание пустого контейнера*
`void`	`add(Item item)`	*добавление элемента*
`boolean`	`isEmpty()`	*пуст ли контейнер?*
`int`	`size()`	*количество элементов в контейнере*

Очередь FIFO

public class **Queue<Item>** implements Iterable<Item>		
	`Queue()`	*создание пустой очереди*
`void`	`enqueue(Item item)`	*добавление элемента*
`Item`	`dequeue()`	*удаление самого "старого" элемента*
`boolean`	`isEmpty()`	*пуста ли очередь?*
`int`	`size()`	*количество элементов в очереди*

Стек LIFO

public class **Stack<Item>** implements Iterable<Item>		
	`Stack()`	*создание пустого стека*
`void`	`push(Item item)`	*добавление элемента*
`Item`	`pop()`	*удаление самого "свежего" элемента*
`boolean`	`isEmpty()`	*пуст ли стек?*
`int`	`size()`	*количество элементов в стеке*

Рис. 1.3.1. *API-интерфейсы для фундаментальных обобщенных итерируемых коллекций*

Обобщения

Важной характеристикой АТД коллекций является то, что мы должны иметь возможность использовать их для любого типа данных. Эту возможность обеспечивает специальный механизм Java, который называется *обобщениями* или *параметризованными типами*. Обобщения имеются не во всех языках программирования (и даже отсутствуют в ранних версиях Java), да и мы будем использовать их здесь лишь в небольшой степени — т.е. лишь небольшое и вполне понятное расширение синтаксиса Java. Запись `<Item>` после имени класса в каждом API-интерфейсе на рис. 1.3.1 определяет имя `Item` как *параметр типа*, т.е. символьный заполнитель для некоего конкретного типа, который будет использовать клиент. Запись `Stack<Item>` можно понимать как "стек элементов". При реализации типа `Stack` мы не знаем конкретный тип `Item`, но клиент может использовать наш стек для любого типа данных, в том числе и определенного через долгое время после разработки реализации. При создании стека клиентский код сообщает конкретный тип: идентификатор `Item` можно заменить именем *любого* ссылочного типа данных (в любом месте, где он может появиться). Это дает как раз то, что нам нужно. Например, можно записать код

```
Stack<String> stack = new Stack<String>();
stack.push("Test");
...
String next = stack.pop();
```

для применения стека с объектами String и код

```
Queue<Date> queue = new Queue<Date>();
queue.enqueue(new Date(12, 31, 1999));
...
Date next = queue.dequeue();
```

для использования очереди с объектами типа Date. При попытке добавить объект Date (или данные любого другого типа, отличного от String) в стек stack или String (или любого другого типа, отличного от Date) в очередь queue возникнет ошибка времени компиляции. Без обобщений пришлось бы определять (и реализовывать) различные API-интерфейсы для каждого типа данных, который может понадобиться в коллекциях, а обобщения позволяют ограничиться одним API-интерфейсом (и одной реализацией) для всех типов данных, даже таких типов, которые еще только будут реализованы. Как вы вскоре убедитесь, обобщенные типы приводят к ясному клиентскому коду, который легко понимать и отлаживать, поэтому мы и будем применять их в книге.

Автоупаковка

На место параметров типа должны подставляться *ссылочные* типы, поэтому в Java имеются специальные механизмы, которые позволяют использовать обобщенный код с примитивными типами. Вы уже знаете, что типы оболочек Java представляют собой ссылочные типы, соответствующие примитивным типам: Boolean, Byte, Character, Double, Float, Integer, Long и Short соответствуют типам boolean, byte, char, double, float, int, long и short. Java автоматически осуществляет преобразования между этими ссылочными типами и соответствующими примитивными типами — в присваиваниях, аргументах методов и арифметических или логических выражениях. В данном контексте такие преобразования позволяют использовать обобщения с примитивными типами, как в следующем коде:

```
Stack<Integer> stack = new Stack<Integer>();
stack.push(17);                // автоупаковка (int -> Integer)
int i = stack.pop();           // автораспаковка (Integer -> int)
```

Автоматическое приведение примитивного типа к типу оболочки называется *автоупаковкой*, а автоматическое приведение типа оболочки к примитивному — *автораспаковкой*. В этом примере Java автоматически выполняет приведение (автоупаковку) примитивного значения 17 к типу Integer при передаче его в метод push(). Метод pop() возвращает значение Integer, который Java приводит (распаковывает) к типу int перед его присваиванием переменной i.

Итерируемые коллекции

Во многих задачах клиенту нужно просто каким-то образом обработать каждый элемент коллекции — т.е. *итерировать* элементы этой коллекции. Эта парадигма настолько важна, что получила приоритетный статус в Java и многих других современных языках: она поддерживается специальными механизмами языка программирования, а не просто библиотеками. С ее помощью можно писать ясный и компактный код, который не зависит от деталей реализации коллекции.

Например, пусть клиент хранит в очереди `Queue` коллекцию транзакций:

```
Queue<Transaction> collection = new Queue<Transaction>();
```

Если коллекция допускает итерацию, клиент может вывести список транзакций с помощью краткой конструкции:

```
for (Transaction t : collection)
{ StdOut.println(t); }
```

Эту конструкцию часто называют оператором *foreach* (for each — для каждого): данный оператор `for` можно прочитать так: ***для каждой транзакции*** `t` ***из коллекции выполнить следующий блок кода.*** Такой клиентский код не должен знать ничего о представлении коллекции — ему нужно просто обработать каждый элемент этой коллекции. Аналогичному циклу `for` пришлось бы работать с контейнером транзакций или с другим видом итерируемой коллекции. Вряд ли можно представить себе более понятный и компактный клиентский код. Как мы увидим, поддержка этой возможности требует дополнительных усилий при реализации, но эти усилия, несомненно, окупаются.

Интересно, что API-интерфейсы для типов `Stack` и `Queue` различаются только именами — самих типов и содержащихся методов. Это наблюдение подчеркивает мысль, что не всегда достаточно просто перечислить все характеристики типа данных с помощью списка сигнатур методов. Правильная спецификация должна содержать описания на "человеческом" языке, задающие правила выбора удаляемого элемента или элемента, который обрабатывается следующим в операторе *foreach*. Различия в этих правилах существенны, они составляют *часть API-интерфейса* и, несомненно, крайне важны для разработки клиентского кода.

Контейнеры

Контейнер (bag) — это коллекция, не поддерживающая удаление элементов; он предназначен для того, чтобы клиенты могли собирать в одном месте элементы и выполнять по ним итерацию (клиент может также проверить, пуст ли контейнер, и узнать количество элементов). Порядок итерации не регламентируется и не должен быть важен клиенту. Для демонстрации этой идеи рассмотрим пример — алчного коллекционера, собирающего шарики. Он складывает шарики один за другим в мешочек и иногда перебирает их, чтобы найти шарик с какими-то особыми свойствами (рис. 1.3.2). API-интерфейс `Bag` позволяет добавлять элементы в мешочек и при необходимости обработать их все с помощью оператора *foreach*. Такой клиент может, конечно, использовать стек или очередь, но использование контейнера подчеркивает безразличие к порядку итерации. Класс `Stats` в листинге 1.3.1 является примером типичного клиента типа `Bag`. Его несложная задача состоит в вычислении среднего значения и среднеквадратичного отклонения значений `double`, принятых из стандартного ввода. Если из стандартного ввода поступило N чисел, то для вычисления их среднего значения необходимо сложить

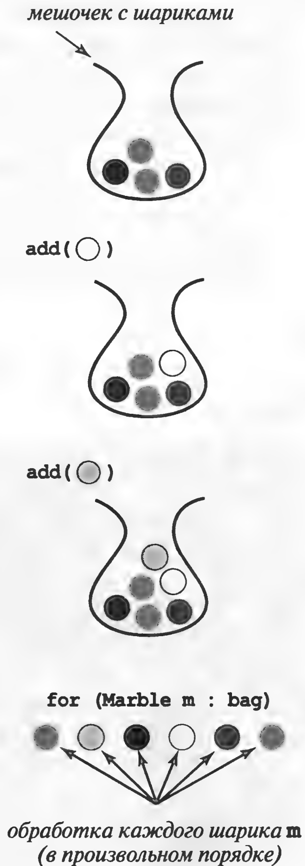


Рис. 1.3.2. Операции с контейнером

все числа и разделить сумму на N, а для вычисления среднеквадратичного отклонения нужно вычислить сумму квадратов разностей каждого числа и среднего значения, разделить на $N-1$ и найти квадратный корень. Порядок итерации чисел никак не влияет на оба эти вычисления, поэтому мы сохраняем их в контейнере и используем конструкцию *foreach* для вычисления обеих сумм. *Примечание*: среднеквадратичное отклонение можно вычислить и без сохранения всех чисел (примерно как и среднее значение в типе `Accumulator` — см. упражнение 1.2.18), но для более сложного статистического анализа потребуется хранить все числа.

Листинг 1.3.1. Типичный клиент класса Bag

```
public class Stats
{
   public static void main(String[] args)
   {
      Bag<Double> numbers = new Bag<Double>();

      while (!StdIn.isEmpty())
         numbers.add(StdIn.readDouble());
      int N = numbers.size();

      double sum = 0.0;
      for (double x : numbers)
         sum += x;
      double mean = sum/N;

      sum = 0.0;
      for (double x : numbers)
         sum += (x - mean)*(x - mean);
      double std = Math.sqrt(sum/(N-1));

      StdOut.printf("Среднее: %.2f\n", mean);
      StdOut.printf("Ср.-кв. откл.: %.2f\n", std);
   }
}
```

Применение

```
% java Stats
100
99
101
120
98
107
109
81
101
90

Среднее: 100.60
Ср.-кв. откл.: 10.51
```

Очереди FIFO

Очередь FIFO (или просто *очередь*) — это коллекция, в которой действует правило *первым вошел — первым вышел* (first in, first out — FIFO). Такое правило выполнения задач в порядке поступления мы часто наблюдаем в повседневной жизни: люди, ожидающие в очереди в театр, машины, ожидающие в очереди на мойку, задачи, ожидающие обслуживания приложением в компьютере. В основе любого правила обслуживания лежит принцип честности: люди считают, что будет честно, если тот, кто ждал дольше всех, обслуживается первым. Это как раз и есть правило FIFO (рис. 1.3.3). Очереди естественно моделируют многие привычные явления и поэтому играют важную роль во многих приложениях. Когда клиент перебирает элементы очереди с помощью конструкции *foreach*, эти элементы выбираются в порядке занесения в очередь. Обычная причина использования очередей в приложениях — сохранение элементов в коллекции *в относительном порядке их поступления*: они выбираются из очереди в том же порядке, в котором заносятся. Например, клиент, приведенный в листинге 1.3.2, представляет собой возможную реализацию статического метода `readInts()` из класса `In`. Этот метод освобождает клиент от решения задачи сохранения в массиве чисел, читаемых из файла, *без предварительного знания размера файла*. Числа из файла *заносятся в очередь* (enqueue), с помощью метода `size()` из класса `Queue` определяется необходимый размер массива, потом создается массив, и затем числа *извлекаются из очереди* (dequeue) для запоминания в массиве. Очередь здесь удобна как раз тем, что числа заносятся в массив в том порядке, в котором они находятся в файле (если этот порядок неважен, можно использовать контейнер). В этом коде выполняются автоупаковка и автораспаковка для преобразования примитивного типа `int` в тип оболочки `Integer` и обратно.

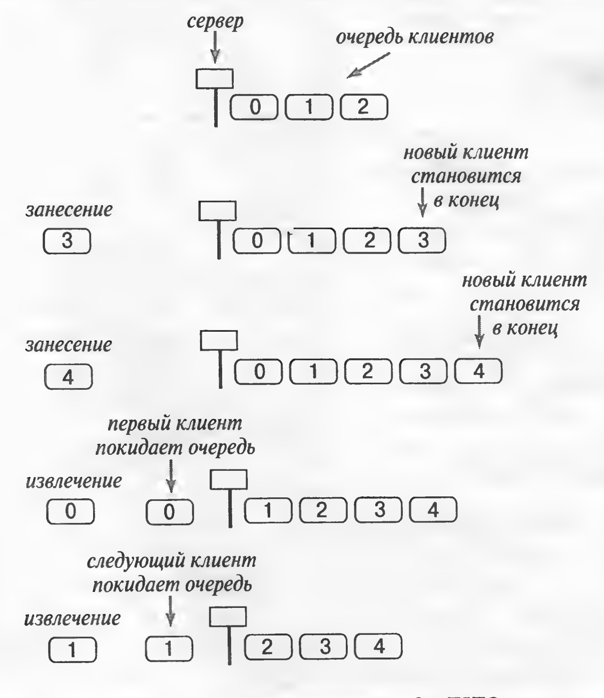


Рис. 1.3.3. Типичная очередь FIFO

Листинг 1.3.2. Пример клиента класса Queue

```
public static int[] readInts(String name)
{
   In in = new In(name);
   Queue<Integer> q = new Queue<Integer>();
   while (!in.isEmpty())
      q.enqueue(in.readInt());

   int N = q.size();
   int[] a = new int[N];
   for (int i = 0; i < N; i++)
      a[i] = q.dequeue();
   return a;
}
```

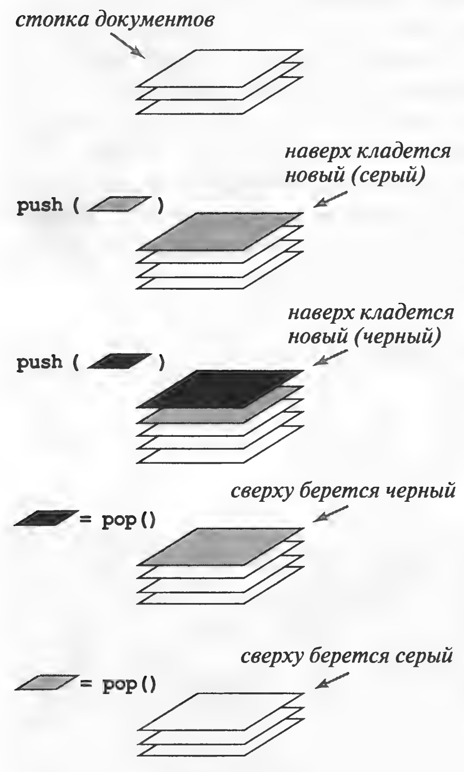


Рис. 1.3.4. Операции со стеком

Стеки LIFO

Стек — это коллекция, в которой действует правило ***первым вошел — последним вышел*** (first in, last out — LIFO). Если вы храните конверты с письмами на столе или полке, то вы пользуетесь стеком. Вы кладете конверты наверх стопки по мере их появления и берете конверты сверху, когда есть время прочитать их. Сейчас бумажная почта используется не так часто, но аналогичные принципы лежат в основе нескольких приложений, которые часто применяются в компьютере. Например, многие используют свою почту в качестве стека: они *заталкивают* (push) сообщения в верхнюю часть списка при их поступлении и *выталкивают* (pop) их сверху при их прочтении — тогда просматриваются в первую очередь самые свежие поступления (рис. 1.3.4). Преимущество этой стратегии в том, что самые интересные сообщения просматриваются максимально быстро, но какое-то старое сообщение может оказаться не просмотренным очень долго. Еще один распространенный пример стека — просмотр информации в Интернете. При щелчке на гиперссылке браузер отображает новую страницу, а старую заталкивает в стек. Можно щелкать на гиперссылках, переходя со страницы на страницу, но всегда можно вернуться на предыдущую страницу, щелкнув на кнопке возврата (т.е. вытолкнув ее из стека). Правило LIFO, по которому работает стек, как раз обеспечивает такое поведение. Когда клиент перебирает элементы стека с помощью конструкции *foreach*, элементы обрабатываются в порядке, *обратном* их поступлению. Обычная причина применения стека в приложениях — сохранение элементов в коллекции с *обращением* их относительного порядка. Например, клиент `Reverse`, приведенный в листинге 1.3.3, изменяет порядок

целых чисел, вводимых из стандартного ввода — и опять без предварительного знания их количества. Стеки играют в компьютерных вычислениях не просто важную, а фундаментальную роль, как мы убедимся в следующем примере.

Листинг 1.3.3. Простой клиент класса Stack

```
public class Reverse
{
  public static void main(String[] args)
  {
    Stack<Integer> stack;
    stack = new Stack<Integer>();
    while (!StdIn.isEmpty())
      stack.push(StdIn.readInt());
    for (int i : stack)
      StdOut.println(i);
  }
}
```

Вычисление арифметического выражения

В качестве еще одного примера клиента стека мы рассмотрим классический пример, которые заодно продемонстрирует пользу обобщений. Одна из первых программ, рассмотренных в разделе 1.1, содержала вычисление арифметических выражений наподобие

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

Если умножить 4 на 5, сложить 2 и 3, перемножить результаты и добавить 1, то получится значение 101. Но как это вычисление выполняет система Java? Можно и не вдаваться в изучение внутренних механизмов Java, а просто написать Java-программу, которая принимает на входе строку (выражение) и выдает число, равное значению этого выражения. Для простоты мы примем следующее явное рекурсивное определение: *арифметическое выражение* — это либо число, либо открывающая скобка, за которой следует арифметическое выражение, символ операции, еще одно арифметическое выражение и закрывающая скобка. Эта формулировка определяет *полностью скобочные* арифметические выражения, где точно указано, какие операции применяются к каким операндам — хотя вы наверняка больше привыкли к выражениям вроде `1 + 2 * 3`, где вместо скобок действуют правила старшинства операций. Механизмы, которые мы рассмотрим, применимы и к правилам старшинства, но мы не будем вдаваться в излишние сложности. А именно: мы будем поддерживать знакомые бинарные операции +, -, * и /, а также операцию извлечения квадратного корня `sqrt`, которая принимает только один аргумент. В этот набор нетрудно добавить другие операции, чтобы охватить большой класс знакомых математических выражений, включая тригонометрические, экспоненциальные и логарифмические функции. Но для нас важнее понять принцип интерпретации строк, состоящих из скобок, знаков операций и чисел, чтобы выполнять в правильном порядке низкоуровневые операции, доступные на любом компьютере. И тогда возникает вопрос: как можно преобразовать арифметическое выражение — т.е. строку символов — в значение, которое оно представляет? Для этого в 1960-х годах Э.У. Дейкстра разработал замечательный алгоритм, использующий два стека: один для операндов, а другой для операций. Выражение состоит из скобок, знаков операций и операндов (чисел).

5 Зак. 3239

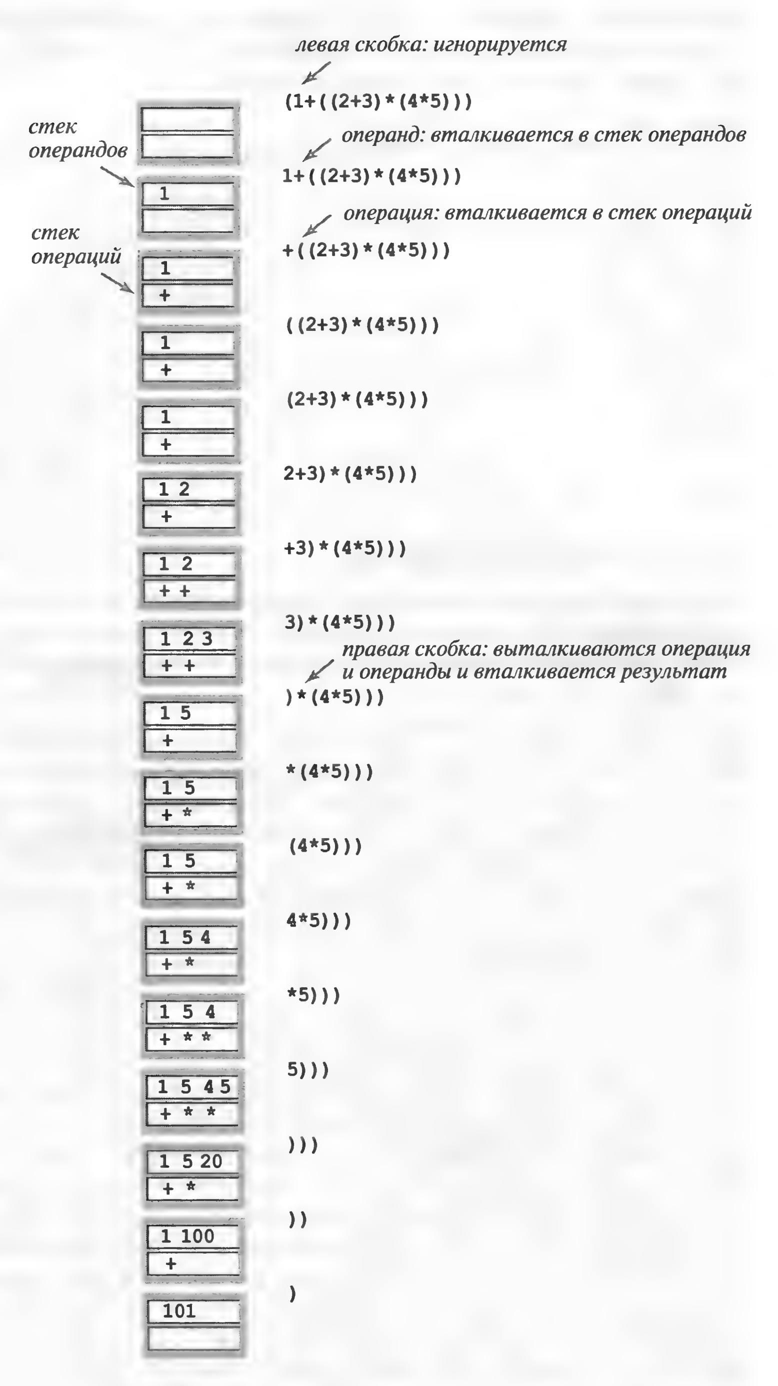


Рис. 1.3.5. Трассировка двухстекового алгоритма Дейкстры для вычисления арифметического выражения

Выбирая эти элементы слева направо, можно использовать стеки в соответствии с четырьмя возможными случаями:

- вталкивать *операнды* в стек операндов;
- вталкивать *операции* в стек операций;
- игнорировать *левые* скобки;
- при достижении *правой* скобки вытолкнуть операцию, вытолкнуть необходимое количество операндов и втолкнуть в стек операндов результат действия операции на эти операнды.

После обработки последней правой скобки в стеке должно остаться только одно значение, которое и является значением выражения (рис. 1.3.5). Этот метод вначале кажется несколько загадочным, но нетрудно убедиться, что он действительно вычисляет нужное значение: каждый раз, когда алгоритм встречает подвыражение, состоящее из двух операндов с операцией между ними и заключенное в скобки, он оставляет в стеке операндов результат выполнения этой операции над этими операндами. Результат такой же, как если бы это выражение сразу было во входной строке вместо подвыражения, и поэтому данное подвыражение можно заменить его значением и получить тот же результат. Это действие можно повторять многократно, пока не останется единственное значение. Например, алгоритм вычисляет одно и то же значение для всех следующих выражений:

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
( 1 + ( 5 * ( 4 * 5 ) ) )
( 1 + ( 5 * 20 ) )
( 1 + 100 )
101
```

Класс `Evaluate`, приведенный в листинге 1.3.4, представляет собой реализацию этого алгоритма. Это простой пример *интерпретатора* — программы, которая выполняет вычисление, заданное некоторой строкой, и получает результат этого вычисления.

Листинг 1.3.4. Двухстековый алгоритм Дейкстры для вычисления выражения

```
public class  Evaluate
{
  public static void main(String[] args)
  {
    Stack<String> ops  = new Stack<String>();
    Stack<Double> vals = new Stack<Double>();
    while (!StdIn.isEmpty())
    {  // Чтение элемента и вталкивание его в стек, если это операция.
       String s = StdIn.readString();
       if      (s.equals("("))              ;
       else if (s.equals("+"))    ops.push(s);
       else if (s.equals("-"))    ops.push(s);
       else if (s.equals("*"))    ops.push(s);
       else if (s.equals("/"))    ops.push(s);
       else if (s.equals("sqrt")) ops.push(s);
       else if (s.equals(")"))
       {  // Если ")", то выталкивание, вычисление и вталкивание результата.
          String op = ops.pop();
          double v = vals.pop();
          if      (op.equals("+"))    v = vals.pop() + v;
          else if (op.equals("-"))    v = vals.pop() - v;
```

```
        else if (op.equals("*"))    v = vals.pop() * v;
        else if (op.equals("/"))    v = vals.pop() / v;
        else if (op.equals("sqrt")) v = Math.sqrt(v);
        vals.push(v);
      }  // Элемент не операция и не скобка: вталкиваем значение double.
      else vals.push(Double.parseDouble(s));
    }
    StdOut.println(vals.pop());
  }
}
```

Приведенный клиент класса Stack использует два стека для вычисления арифметических выражений и демонстрирует важный вычислительный процесс — интерпретацию строки как программы и выполнение этой программы для вычисления нужного результата. Обобщения позволяют использовать единственный класс Stack для реализации одного стека для значений String и другого — для значений Double. Для простоты здесь предполагается, что значение является полностью скобочным, а числа и символы разделены пробельными символами.

```
% java Evaluate
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
101.0
% java Evaluate
( ( 1 + sqrt ( 5.0 ) ) / 2.0 )
1.618033988749895
```

Реализация коллекций

Сейчас мы приступим к реализации классов Bag, Stack и Queue и начнем с простой классической реализации, а затем рассмотрим усовершенствования, которые приведут к реализациям API-интерфейсам, приведенным на рис. 1.3.1.

Стек фиксированной емкости

В качестве первоначального варианта мы рассмотрим абстрактный тип данных для стека строк, имеющего фиксированную емкость, как показано на рис. 1.3.6. Этот API-интерфейс отличается от нашего API-интерфейс для класса Stack: он работает только со значениями типа String, требует, чтобы клиент указал емкость, и не поддерживает итерацию. Главная цель нашей разработки такой реализации — *выбрать представление данных*. Для типа FixedCapacityStackOfStrings естественным будет выбор массива значений String. Этот выбор приведет нас к максимально простой реализации, показанной в нижней части рис. 1.3.6: каждый метод умещается в одну строку. Переменные экземпляров — массив a[], который содержит значения стека, и целое число N, содержащее количество элементов в стеке. Для удаления (выталкивания) элемента нужно уменьшить на единицу N и возвратить a[N]; для вставки (вталкивания) нового элемента нужно занести этот новый элемент в a[N], а затем увеличить на единицу N. Эти операции обладают следующими свойствами:

- элементы находятся в массиве в порядке их вставки;
- стек пуст, когда N равно 0;
- вершина (непустого) стека находится в элементе a[N-1].

API-интерфейс

public class **FixedCapacityStackOfStrings**		
	FixedCapacityStackOfStrings(int cap)	*создание пустого стека емкостью* cap
void	push(String item)	*добавление строки*
String	pop()	*удаление последней добавленной строки*
boolean	isEmpty()	*пуст ли стек?*
int	size()	*количество строк в стеке*

Клиент тестирования

```
public static void main(String[] args)
{
   FixedCapacityStackOfStrings s;
   s = new FixedCapacityStackOfStrings(100);
   while (!StdIn.isEmpty())
   {
      String item = StdIn.readString();
      if (!item.equals("-"))
          s.push(item);
      else if (!s.isEmpty()) StdOut.print(s.pop() + " ");
   }
   StdOut.println("(в стеке осталось " + s.size() + ")");
}
```

Применение

```
% more tobe.txt
to be or not to - be - - that - - - is
% java FixedCapacityStackOfStrings < tobe.txt
to be not that or be (в стеке осталось 2)
```

Реализация

```
public class FixedCapacityStackOfStrings
{
   private String[] a;          // элементы стека
   private int N;               // размер
   public FixedCapacityStackOfStrings(int cap)
   { a = new String[cap]; }
   public boolean isEmpty() { return N == 0; }
   public int size()        { return N; }
   public void push(String item)
   { a[N++] = item; }
   public String pop()
   { return a[--N]; }
}
```

Рис. 1.3.6. Абстрактный тип данных для стека строк фиксированной емкости

Мышление такими инвариантами — самый простой способ проверить правильность работы реализации. *Вы должны полностью понимать эту реализацию*. А этого проще всего добиться, рассматривая трассировку содержимого стека на протяжении ряда операций. Такая трассировка приведена в табл. 1.3.1 для клиента тестирования, который читает строки из стандартного ввода и вталкивает их в стек, но при вводе строки "-" осуществляется выталкивание строки из стека и вывод результата. Основная характеристика производительности данной реализации такова: *операции вталкивания и выталкивания выполняются за время, не зависящее от размера стека*. Во многих случаях этот метод оптимален в силу своей простоты. Однако у него есть и недостатки, которые ограничивают его потенциальную применимость в качестве универсального инструмента, и которые мы сейчас рассмотрим. Ценой небольших усилий (и с некоторой помощью механизмов языка Java) можно разработать полезную во многих ситуациях реализацию. Эти усилия не пропадут даром: разрабатываемые нами реализации послужат моделью для реализаций других, более мощных, абстрактных типов данных в этой книге.

Таблица 1.3.1. Трассировка клиента тестирования для класса `FixedCapacityStackOfStrings`

StdIn (вталкивание)	StdOut (выталкивание)	N	a[]				
			0	1	2	3	4
		0					
to		1	to				
be		2	to	be			
or		3	to	be	or		
not		4	to	be	or	not	
to		5	to	be	or	not	to
-	to	4	to	be	or	not	to
be		5	to	be	or	not	be
-	be	4	to	be	or	not	be
-	not	3	to	be	or	not	be
that		4	to	be	or	that	be
-	that	3	to	be	or	that	be
-	or	2	to	be	or	that	be
-	be	1	to	be	or	that	be
is		2	to	is	or	not	to

Обобщения

Первый недостаток типа `FixedCapacityStackOfStrings` состоит в том, что он работает только с объектами `String`. Если нам понадобится стек значений `double`, придется разработать другой класс с похожим кодом и везде заменить `String` на `double`. Это вроде и нетрудно, но становится обременительным, если потом понадобится стек значений `Transaction` или очередь значений `Date` и т.п. Как было сказано в начале этого раздела, параметризованные типы (обобщения) в Java специально созданы для разрешения такой ситуации, и мы уже видели несколько примеров соответствующего клиент-

ского кода (листинги 1.3.1—1.3.4). Но как *реализовать* обобщенный стек? Подробности можно увидеть на рис. 1.3.7. На нем приведена реализация класса `FixedCapacityStack`, отличия которой от версии `FixedCapacityStackOfStrings` выделены: все вхождения `String` заменены на `Item` (с одним исключением, о котором будет сказано ниже), и объявление класса в первой строке имеет вид

```
public class FixedCapacityStack<Item>
```

Идентификатор `Item` — это *параметр типа*, т.е. заменитель некоторого конкретного типа, который будет указан клиентом. Запись `FixedCapacityStack<Item>` можно понимать как *стек элементов*, а это как раз то, что нам нужно. При реализации `FixedCapacityStack` мы не знаем реальный тип `Item`, но клиент может использовать данный класс для любого типа данных, указав конкретный тип при создании такого стека. Конкретные типы должны быть ссылочными типами, но клиент может полагаться на автоупаковку для преобразования примитивных типов в соответствующие типы оболочек. Java использует параметр типа `Item` для проверки на наличие несовпадений типов — хотя конкретный тип еще неизвестен, переменные типа `Item` должны присваиваться переменным типа `Item` и т.п. Но во всей этой истории есть один примечательный момент: в конструкторе `FixedCapacityStack` хорошо бы использовать оператор

```
a = new Item[cap];
```

и таким образом создать обобщенный массив. По историческим и техническим причинам, в которые мы не будем вдаваться, *обобщенные массивы в Java не разрешены*. Вместо этого необходимо выполнить приведение:

```
a = (Item[]) new Object[cap];
```

Этот код делает именно то, что нам нужно (хотя компилятор Java выдаст предупреждение, которое можно спокойно проигнорировать), и мы будем использовать эту идиому далее во всей книге (она применяется и в реализациях аналогичных абстрактных типов данных в системных библиотеках Java).

Изменение размера массива

Выбор массива для хранения содержимого стека означает, что клиенты должны заранее оценивать максимальный размер стека. Java не допускает изменение размера массива после его создания, поэтому стек всегда использует объем памяти, пропорциональный этому максимуму. Клиент, выбравший большой объем, рискует потратить зря большой участок памяти тогда, когда коллекция пуста или почти пуста. Например, система обработки транзакций может работать с миллиардами элементов и тысячами коллекций таких элементов. В таком клиенте должна быть предусмотрена возможность, что любая из этих коллекций будет содержать все элементы, хотя обычно каждый элемент может принадлежать лишь одной коллекции. И каждый клиент рискует столкнуться с *переполнением*, если размер коллекции превысит размер массива. Поэтому в метод `push()` необходимо добавить код проверки, полон ли стек, а в API-интерфейс добавить метод `isFull()` для выполнения такой проверки извне. Мы не будем приводить этот код, т.к. не хотим грузить клиент дополнительной концепцией заполненного стека, как и сформулировано в нашем первоначальном API-интерфейсе `Stack`. Вместо этого мы добавим в реализацию посредством массива динамическое изменение размера массива `a[]`, чтобы массив был достаточно большим для хранения всех элементов, но не слишком большим, чтобы не тратить зря большой объем памяти. Оказывается, это совсем нетрудно сделать.

API-интерфейс

public class **FixedCapacityStack<Item>**		
	FixedCapacityStack(int cap)	*создание пустого стека емкостью* cap
void	push(**Item** item)	*добавление элемента*
Item	pop()	*удаление последнего добавленного элемента*
boolean	isEmpty()	*пуст ли стек?*
int	size()	*количество элементов в стеке*

Клиент тестирования

```
public static void main(String[] args)
{
   FixedCapacityStack<String> s;
   s = new FixedCapacityStack<String>(100);
   while (!StdIn.isEmpty())
   {
      String item = StdIn.readString();
      if (!item.equals("-"))
          s.push(item);
      else if (!s.isEmpty()) StdOut.print(s.pop() + " ");
   }
   StdOut.println("(в стеке осталось " + s.size() + ")");
}
```

Применение

```
% more tobe.txt
to be or not to - be - - that - - - is

% java FixedCapacityStackOfStrings < tobe.txt
to be not that or be (в стеке осталось 2)
```

Реализация

```
public class FixedCapacityStack<Item>
{
   private Item[] a;            // элементы стека
   private int N;               // размер
   public FixedCapacityStackOfStrings(int cap)
   { a = (Item[]) new Object[cap]; }
   public boolean isEmpty() { return N == 0; }
   public int size()        { return N; }
   public void push(Item item)
   { a[N++] = item; }
   public Item pop()
   { return a[--N]; }
}
```

Рис. 1.3.7. Абстрактный тип данных для обобщенного стека фиксированной емкости

Вначале мы реализуем метод, который перемещает стек в массив другого размера:

```
private void resize(int max)
{ // Перенос стека размером N <= max в новый массив размером max.
  Item[] temp = (Item[]) new Object[max];
  for (int i = 0; i < N; i++)
    temp[i] = a[i];
  a = temp;
}
```

Теперь в метод `push()` нужно добавить проверку, исчерпана ли емкость массива. А именно, мы проверяем, имеется ли в массиве место для нового элемента, проверяя, равен ли размер стека `N` размеру массива `a.length`. Если места уже нет, мы ***увеличим вдвое*** размер массива, а потом, как и раньше, вставим новый элемент с помощью операции `a[N++] = item`:

```
public void push(String item)
{ // Добавление элемента на верхушку стека.
  if (N == a.length) resize(2*a.length);
  a[N++] = item;
}
```

Аналогично и в методе `pop()`: сначала мы удаляем элемент, а затем ***уменьшаем вдвое*** размер массива, если он слишком велик. Удобно выполнять проверку, меньше ли размер стека, чем ***четверть*** размера массива. Тогда после уменьшения массива он будет заполнен примерно наполовину и сможет обслужить немало операций `push()` и `pop()`, прежде чем появится необходимость снова изменить размер массива.

```
public String pop()
{ // Удаление элемента с верхушки стека.
  String item = a[--N];
  a[N] = null;  // Гашение праздной ссылки (см. текст).
  if (N > 0 && N == a.length/4) resize(a.length/2);
  return item;
}
```

При такой реализации стек никогда не переполнится и никогда не будет заполнен менее чем на четверть (кроме случая пустого стека, когда размер массива равен 1). Анализ производительности этого подхода будет приведен в разделе 1.4.

Бесхозные ссылки

В Java выполняется сборка мусора, т.е. возврат памяти, связанной с объектами, к которым уже невозможно обратиться. В нашей реализации метода `pop()` ссылка на вытолкнутый элемент остается в массиве. Сам элемент является *висячим* — ведь к нему уже не будет обращений — но сборщик мусора не узнает об этом до момента перезаписи ссылки. Даже когда клиент прекратит работу с элементом, ссылка на него в массиве останется рабочей. Такая ситуация (хранение ссылки на уже не нужный элемент) называется *бесхозной ссылкой*. В нашем случае ее легко избежать: достаточно в элемент массива, соответствующий вытолкнутому элементу стека, занести `null`. Такая перезапись ненужной ссылки позволяет системе вернуть память, связанную с вытолкнутым элементом, когда клиент закончит работу с ним (см. табл. 1.3.2).

Таблица 1.3.2. Трассировка изменения размера массива при выполнении последовательности операций push() и pop()

push()	pop()	N	a.length	a[]							
				0	1	2	3	4	5	6	7
		0	1	*null*							
to		1	1	to							
be		2	2	to	be						
or		3	4	to	be	or	*null*				
not		4	4	to	be	or	not				
to		5	8	to	be	or	not	to	*null*	*null*	*null*
-	to	4	8	to	be	or	not	*null*	*null*	*null*	*null*
be		5	8	to	be	or	not	be	*null*	*null*	*null*
-	be	4	8	to	be	or	not	*null*	*null*	*null*	*null*
-	not	3	8	to	be	or	*null*	*null*	*null*	*null*	*null*
that		4	8	to	be	or	that	*null*	*null*	*null*	*null*
-	that	3	8	to	be	or	*null*	*null*	*null*	*null*	*null*
-	or	2	4	to	be	*null*	*null*				
-	be	1	2	to	*null*						
is		2		to	is						

Итерация

Как уже было сказано в настоящем разделе, одной из фундаментальных операций с коллекциями является обработка каждого элемента во время *итерации* по коллекции оператором *foreach*. Эта парадигма приводит к ясному и компактному коду, который не зависит от незначащих деталей реализации коллекции. Рассматривать задачу реализации итерации мы начнем с фрагмента клиентского кода, который выводит все элементы коллекции строк, по одной в строке вывода:

```
Stack<String> collection = new Stack<String>();
...
for (String s : collection)
   StdOut.println(s);
...
```

Приведенный здесь оператор for является сокращением для конструкции while, и поэтому данный код примерно эквивалентен следующему коду:

```
Iterator<String> i = collection.iterator();
while (i.hasNext())
{
   String s = i.next();
   StdOut.println(s);
}
```

В этом коде уже хорошо видны все ингредиенты, которые необходимы для реализации любой итерируемой коллекции:

- коллекция должна реализовывать метод `iterator()`, который возвращает объект `Iterator`;
- класс `Iterator` должен содержать два метода: `hasNext()` (возвращающий логическое значение) и `next()` (возвращающий обобщенный элемент из коллекции).

Для выражения требования, что некоторый класс должен реализовывать какой-то метод, в Java имеется механизм интерфейсов (см. подраздел “Наследование интерфейса” в разделе 1.2). Для итерируемых коллекций Java определяет за нас все необходимые интерфейсы. Чтобы сделать класс итерируемым, необходимо добавить в начало его объявления конструкцию `implements Iterable<Item>`, которая означatn, что этот класс должен соответствовать интерфейсу

```
public interface Iterable<Item>
{
   Iterator<Item> iterator();
}
```

(находится в библиотеке `java.lang.Iterable`), а затем добавить в класс метод `iterator()`, возвращающий объект `Iterator<Item>`. Итераторы представляют собой обобщенные методы, и это позволяет использовать параметризованный тип `Item` для итерации по объектам любого типа, который предоставит клиент. Для рассматриваемого сейчас представления в виде массива нужно выполнять просмотр элементов массива в обратном порядке, поэтому мы назовем наш итератор `ReverseArrayIterator` и добавим в класс такой метод:

```
public Iterator<Item> iterator()
{ return new ReverseArrayIterator(); }
```

Что такое итератор? Это объект класса, в котором реализованы методы `hasNext()` и `next()`, как указано в следующем интерфейсе (из библиотеки `java.util.Iterator`):

```
public interface Iterator<Item>
{
    boolean hasNext();
    Item next();
    void remove();
}
```

В интерфейсе задан и метод `remove()`, но мы в данной книге всегда будем указывать для него пустой метод, поскольку совместного использования итерации и операций, изменяющих структуру данных, лучше не допускать. Для класса `ReverseArrayIterator` эти методы умещаются в одну строку и реализуются во вложенном классе нашего класса стека:

```
private class ReverseArrayIterator implements Iterator<Item>
{
   private int i = N;
   public boolean hasNext() { return i > 0;  }
   public Item next()       { return a[--i]; }
   public void remove()     {                }
}
```

Этот вложенный класс имеет доступ к переменным экземпляров объемлющего класса — в данном случае a[] и N (эта возможность и является основной причиной, по которой мы используем для итераторов вложенные классы). Формально, чтобы удовлетворить спецификации Iterator, следовало бы генерировать исключения в двух случаях: UnsupportedOperationException, если клиент вызвал метод remove(), и NoSuchElementException, если клиент вызвал метод next(), когда i равно 0. Поскольку итераторы применяются только в конструкции *foreach*, где такие условия не могут возникнуть, мы опускаем этот код. Но один важный нюанс остается: в начало программы понадобится добавить оператор

```
import java.util.Iterator;
```

т.к. (по историческим причинам) класс Iterator не принадлежит пространству java.lang (хотя Iterable *принадлежит*). Теперь клиент, использующий оператор *foreach* для этого класса, обретет поведение, эквивалентное обычному циклу for для массивов, но ему не надо вникать в строение этих массивов (детали реализации). Такая организация очень важна для реализаций таких фундаментальных типов данных, как коллекции, которые мы рассматриваем в книге и которые включены в Java-библиотеки. Например, мы можем свободно перейти к совершенно другому представлению *без необходимости изменения клиентского кода*. А с точки зрения клиента важнее то, что при этом клиенты могут использовать итерацию, *не зная никаких деталей реализации класса*.

Алгоритм 1.1, приведенный в листинге 1.3.5, содержит реализацию нашего API-интерфейса Stack, который изменяет размер массива, позволяет клиентам создавать стеки для любых типов данных и поддерживает применение в клиентах операторов *foreach* для итерации по элементам массива в соответствии с правилом LIFO. В этой реализации задействованы тонкие моменты Java (Iterator и Iterable), но нет необходимости подробно изучать эти нюансы, поскольку сам код несложен и может служить шаблоном для реализаций других коллекций.

Листинг 1.3.5. Алгоритм 1.1. Стек (реализация с переменным размером массива)

```
import java.util.Iterator;
public class  ResizingArrayStack<Item> implements Iterable<Item>
{
   private Item[] a = (Item[]) new Object[1];         // элементы стека
   private int N = 0;                                  // количество элементов

   public boolean isEmpty()  { return N == 0; }
   public int size()         { return N;      }

   private void resize(int max)
   {  // Перенос стека в новый массив размером max.
      Item[] temp = (Item[]) new Object[max];
      for (int i = 0; i < N; i++)
         temp[i] = a[i];
      a = temp;
   }

   public void push(Item item)
   {  // Добавление элемента на верхушку стека.
      if (N == a.length) resize(2*a.length);
      a[N++] = item;
   }
```

```
    public Item pop()
    {  // Удаление элемента с верхушки стека.
       Item item = a[--N];
       a[N] = null;   // см. текст
       if (N > 0 && N == a.length/4) resize(a.length/2);
       return item;
    }

    public Iterator<Item> iterator()
     { return new ReverseArrayIterator(); }

    private class ReverseArrayIterator implements Iterator<Item>
    {  // Поддержка итерации по правилу LIFO.
       private int i = N;
       public boolean hasNext() { return i > 0;   }
       public    Item next()     { return a[--i]; }
       public    void remove()   {                }
    }
}
```

Эта обобщенная реализация API-интерфейса Stack с возможностью итерации — образец для АТД коллекций, которые хранят элементы в массиве. Размер массива изменяется так, чтобы он оставался примерно пропорциональным размеру стека.

Например, можно реализовать API-интерфейс Queue — с помощью двух индексов в виде переменных экземпляров: переменная head для начала очереди и переменная tail для конца очереди. Чтобы удалить элемент, нужно извлечь его с помощью указателя head, а затем увеличить значение head; чтобы вставить элемент, нужно занести его по указателю tail, а затем увеличить значение tail. Если увеличение индекса вывело его за конец массива, его нужно обнулить (см. табл. 1.3.3). Разработка проверок, когда очередь пуста и когда необходимо изменить размер массива, представляет собой интересное и полезное упражнение по программированию (см. упражнение 1.3.14).

Таблица 1.3.3. Трассировка клиента тестирования для класса ResizingArrayQueue

StdIn (занесение)	StdOut (удаление)	N	head	tail	a[]							
					0	1	2	3	4	5	6	7
		5	0	5	to	be	or	not	to			
-	to	4	1	5	to	be	or	not	to			
be		5	1	6	to	be	or	not	to	be		
-	be	4	2	6	to	be	or	not	to	be		
-	or	3	2	6	to	be	or	not	to	be		

В контексте изучения алгоритмов приведенный алгоритм 1.1 важен тем, что он почти (но не полностью) достигает оптимальной производительности для любой реализации коллекции:

- каждая операция выполняется за время, не зависящее от размера коллекции;
- объем занимаемой памяти примерно пропорционален размеру коллекции.

Недостатком класса ResizingArrayStack можно считать то, что некоторые операции вталкивания и выталкивания требуют изменения размера массива, а на это тратится время, пропорциональное размеру стека. А сейчас мы рассмотрим способ устранения этого недостатка с помощью совершенно другого способа структурирования данных.

Связные списки

Теперь мы рассмотрим фундаментальную структуру данных, которая удобна для представления данных в реализации АТД коллекции. Это будет наш первый опыт построения структуры данных, которая не имеет непосредственной поддержки в языке Java. Наша реализация послужит образцом для кода, который мы будем использовать в этой книге для создания более сложных структур данных — поэтому тщательно проработайте данный раздел, даже если у вас есть опыт работы со связными списками.

Определение. *Связный список* — это рекурсивная структура данных, которая либо пуста (*null*), либо представляет собой ссылку на *узел*, который содержит обобщенный элемент и ссылку на связный список.

Узел в этом определении — некая абстрактная сущность, которая может содержать произвольные данные и ссылку на другой узел, и которая выполняет ключевую роль в построении связных списков. Как и в случае рекурсивных программ, концепция рекурсивной структуры данных поначалу может оказаться слишком трудной для понимания, но она весьма ценна в силу своей простоты.

Запись узла

В объектно-ориентированном программировании реализация связных списков не вызывает сложностей. Вначале мы запишем *вложенный класс*, который определяет абстракцию узла:

```
private class Node
{
   Item item;
   Node next;
}
```

Класс Node содержит две переменных экземпляра типа Item (параметризованный тип) и Node. Этот класс определяется в том классе, где предполагается его использовать, с квалификатором private — поскольку он не предназначен для клиентов. Как и любой другой тип данных, объект типа Node создается с помощью вызова конструктора new Node() без аргументов. В результате создается ссылка на объект Node, в котором обе переменных экземпляра инициализированы значением null. Переменная item представляет собой заполнитель для любого типа данных, которые понадобилось структурировать в виде связного списка (мы задействуем механизм обобщенных типов Java, чтобы он представлял любой ссылочный тип); а переменная типа Node характеризует связную природу структуры данных. Чтобы подчеркнуть, что класс Node предназначен только для структурирования данных, мы не определяем методы, а в коде обращаемся непосредственно к переменным экземпляров: если first — переменная, связанная с объектом типа Node, то мы обращаемся в коде к переменным экземпляров с помощью записи first.item и first.next. Подобные классы иногда называются *записями*. Они не определяют абстрактные типы данных, т.к. мы обращаемся непосредственно к переменным экземпляров. Однако во всех наших реализациях тип Node и его клиентский

код находятся в другом, объемлющем, классе и не доступны клиентам этого класса, т.е. мы все-таки пользуемся преимуществами абстракции данных.

Создание связного списка

Теперь на основе рекурсивного определения мы можем представить связный список переменной типа Node — необходимо просто следить, чтобы ее поле next содержало или *null*, или ссылку на узел Node, поле next которого является ссылкой на связный список. Например, чтобы создать связный список, содержащий строки to, be и or, вначале понадобится создать экземпляры Node для каждого элемента:

```
Node first  = new Node();
Node second = new Node();
Node third  = new Node();
```

потом занести в поле item каждого узла нужное значение (здесь предполагается, что тип Item означает String):

```
first.item  = "to";
second.item = "be";
third.item  = "or";
```

и затем заполнить поля next, чтобы собрать из отдельных элементов связный список:

```
first.next  = second;
second.next = third;
```

(В third.next остается значение null, которое занесено во время создания экземпляра.) Итак, что у нас получилось? Переменная third представляет собой связный список, т.к. это ссылка на узел, который содержит ссылку null — т.е. нулевую ссылку на пустой связный список. Переменная second представляет собой связный список, поскольку это ссылка на узел, который содержит ссылку на связный список third. И переменная first также представляет собой связный список, потому что это ссылка на узел, который содержит ссылку на связный список second. Эти операторы присваивания представлены на рис. 1.3.8, правда, в другом порядке.

Связный список представляет последовательность элементов. В рассмотренном выше примере переменная first представляет последовательность to be or. Такую последовательность можно оформить и в виде массива, например:

```
String[] s = { "to", "be", "or" };
```

Однако в связных списках легче выполнить вставку элемента в последовательность и удаление элемента из последовательности. И сейчас мы рассмотрим код, который выполняет эти задачи.

```
Node first = new Node();
first.item = "to";
```

first: to | null

```
Node second = new Node();
second.item = "be";
first.next  = second;
```

first: to → second: be | null

```
Node third  = new Node();
third.item  = "or";
second.next = third;
```

first: to → second: be → third: or | null

Рис. 1.3.8. Сборка списка из элементов

При трассировке кода, в котором используются связные списки и другие связные структуры, мы будем применять визуальное представление, в котором:

- объекты представляются прямоугольниками;
- значения переменных экземпляров записываются внутри этих прямоугольников;
- ссылки на объекты обозначаются стрелками.

Такое визуальное представление содержит все важные характеристики связных списков. Для краткости мы будем называть ссылки на узлы просто *ссылками* (иногда их называют *связями*). Для простоты, если значения элементов являются строками (как в наших примерах), мы помещаем эти строки непосредственно в прямоугольники объектов, хотя строковые объекты и символьные массивы, рассмотренные в разделе 1.2, точнее было бы изображать отдельно. Это визуальное представление позволит сконцентрироваться на работе со ссылками.

Вставка в начало

Допустим, что в связный список требуется вставить новый узел. Проще всего это сделать в начале списка. Например, для вставки строки `not` в начало списка, в котором первый узел хранит строку `first`, нужно сохранить ссылку `first` в переменной `oldfirst`, занести в `first` ссылку на новый `Node`, и затем занести в поле `item` ссылку на строку `not`, а в поле `next` — значение `oldfirst` (см. также рис. 1.3.9). Этот код вставки узла в начало связного списка требует выполнения лишь нескольких операторов присваивания, поэтому необходимое для этого время не зависит от длины списка.

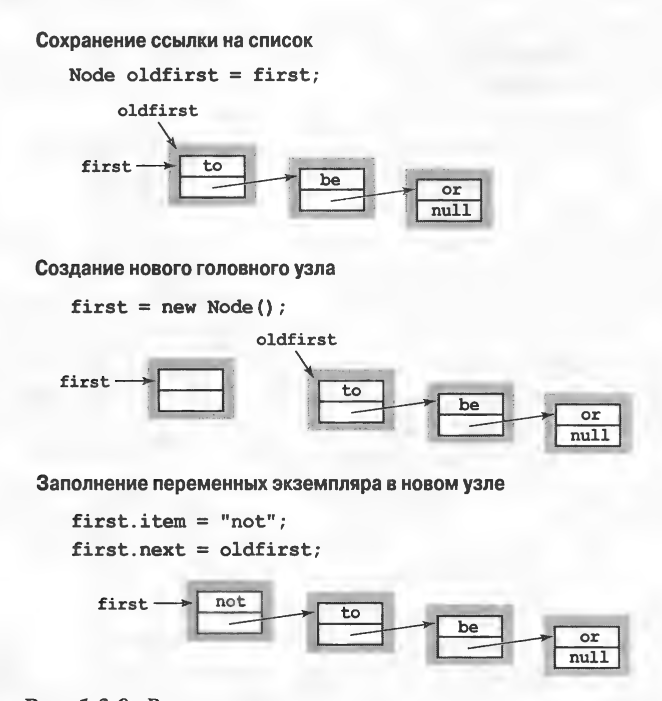


Рис. 1.3.9. Вставка нового узла в начало связного списка

Удаление из начала

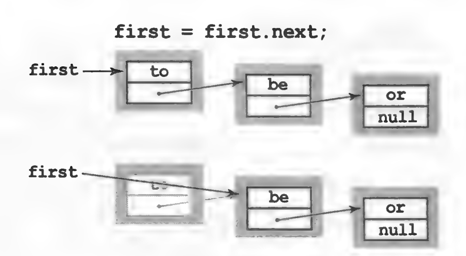


Рис. 1.3.10. Удаление первого узла из связного списка

А теперь предположим, что нужно удалить первый узел из списка. Это сделать даже проще: достаточно присвоить переменной `first` значение `first.next` (рис. 1.3.10). Обычно перед этим производится выборка значения элемента (т.е. присваивание этого значения какой-то переменной типа `Item`), т.к. после изменения значения `first` доступ к первому узлу может оказаться невозможным. Объект бывшего первого узла становится висячим, и система управления памятью в Java в какой-то момент возвращает занимаемую узлом память в пул доступной памяти. Эта операция требует выполнения лишь одного оператора присваивания, и поэтому время ее выполнения также не зависит от длины списка.

Вставка в конец

А как добавить узел в *конец* связного списка? Для этого нужна ссылка на последний узел списка, т.к. в этом узле необходимо изменить ссылку, чтобы она указывала на новый узел со вставленным элементом (рис. 1.3.11). Использование дополнительной ссылки — не такое уж и простое занятие при работе со связными списками, т.к. каждый метод, изменяющий список, должен проверять, нужно ли изменить и эту переменную (и при необходимости изменить). Например, приведенный выше код для удаления первого узла из списка может изменять и ссылку на последний узел: ведь в случае списка только из одного узла этот узел является сразу и первым, и последним. Кроме того, данный метод не работает (переход по пустой ссылке) в случае пустого списка. Такие мелкие детали затрудняют отладку кода работы со связными списками.

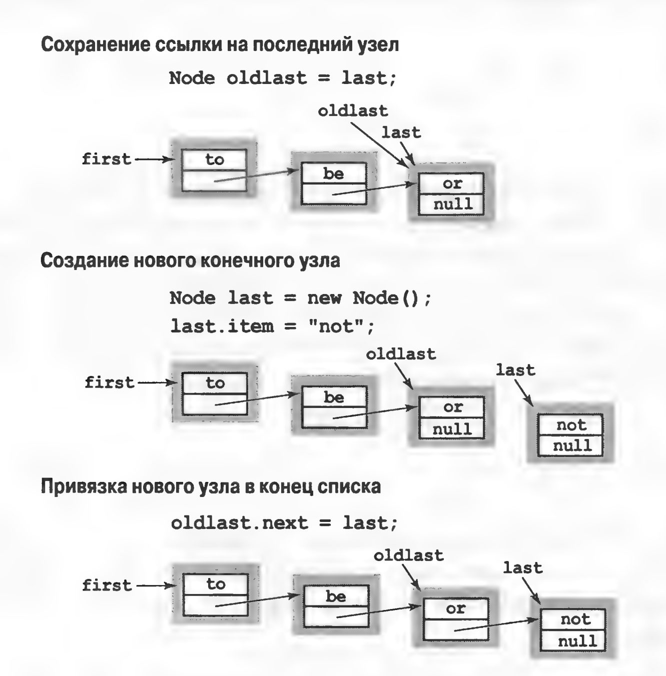


Рис. 1.3.11. Вставка нового узла в конец связного списка

Вставка/удаление в других позициях

Итак, мы продемонстрировали, что мы можем реализовать некоторые операции со связными списками с помощью лишь нескольких инструкций — если есть доступ к ссылкам `first` на первый элемент списка и `last` на последний элемент:

- вставка в начало;
- удаление из начала;
- вставка в конец.

Однако другие операции выполнить не так легко, например:

- удаление заданного узла;
- вставка нового узла перед заданным.

К примеру, как удалить из списка последний узел? Ссылка `last` бесполезна, т.к. нужно заменить значением `null` ссылку в предпоследнем узле списка (которая имеет то же значение, что и `last`). В отсутствие другой информации единственным решением будет проход по всему списку и поиск узла со ссылкой на `last` (см. ниже и упражнение 1.3.19). Такое решение нежелательно, т.к. требует времени, пропорционального длине списка. Стандартное решение, позволяющее выполнять произвольные удаления и вставки — использование *двухсвязных списков*, где каждый узел содержит две ссылки, по одной в каждом направлении. Кодирование этих операций мы оставляем в качестве упражнения (см. упражнение 1.3.31). В наших реализациях двухсвязные списки не понадобятся.

Проход

Для просмотра и обработки каждого элемента массива используется хорошо знакомый код цикла:

```
for (int i = 0; i < N; i++)
{
   // Обработка a[i].
}
```

Соответствующая идиома имеется и для просмотра элементов связного списка. Вначале указатель `x` инициализируется ссылкой на первый узел списка. Теперь мы можем обратиться к элементу, связанному с `x` (это `x.item`), а затем перейти к следующему элементу — для этого понадобится занести в `x` значение `x.next`. Это действие повторяется, пока `x` не станет равным `null`, что означает конец списка. Такой процесс называется ***проходом*** по списку и кратко записывается с помощью кода вроде приведенного ниже цикла, где на первый элемент указывает переменная `first`:

```
for (Node x = first; x != null; x = x.next)
{
   // Обработка x.item.
}
```

Эта идиома настолько же стандартна, как и итерация по элементам массива. В наших реализациях мы будем применять ее в качестве основы для итераторов, позволяющих клиентскому коду перебирать элементы связного списка, не зная деталей реализации этого списка.

Реализация стека

Теперь все готово для разработки реализации для нашего API-интерфейса `Stack` — она приведена в алгоритме 1.2 (листинг 1.3.6). В этой реализации для хранения стека используется связный список, начало которого соответствует верхушке стека, и на него указывает переменная экземпляров `first`. Для вталкивания элемент добавляется в начало списка (как на рис. 1.3.9), а для выталкивания элемент удаляется из начала списка (как на рис. 1.3.10). Для реализации функции `size()` отслеживается количество элементов в переменной экземпляров `N`: значение `N` увеличивается на единицу при вталкивании и уменьшается при выталкивании. В реализации метода `isEmpty()` осуществляется проверка значения `first` на равенство `null` (можно также проверять `N` на равенство `0`). В реализации используется обобщенный тип `Item`: конструкцию `<Item>` после имени класса можно рассматривать как указание, что все вхождения `Item` в реализации будут заменены именем типа данных, указанным клиентом (см. подраздел "Обобщения" выше в данном разделе). Пока мы не включаем код итерации — он будет рассмотрен в листинге 1.3.10. Трассировка клиента тестирования приведена на рис. 1.3.12. Такое применение связного списка дает оптимальные показатели производительности:

- он применим к любому типу данных;
- объем требуемой памяти всегда пропорционален размеру коллекции;
- время выполнения любой операции не зависит от размера коллекции.

Листинг 1.3.6. Алгоритм 1.2. Стек (реализация на основе связного списка)

```
public class  Stack<Item> implements Iterable<Item>
{
  private Node first;       // верхушка стека (узел, добавленный последним)
  private int N;            // количество элементов
  private  class Node
  { // вложенный класс для определения узлов
    Item item;
    Node next;
  }
  public boolean isEmpty() { return first == null; }  // Или N == 0.
  public int size()        { return N; }
  public void push(Item item)
  { // Добавление элемента на верхушку стека.
    Node oldfirst = first;
    first = new Node();
    first.item = item;
    first.next = oldfirst;
    N++;
  }
  public Item pop()
  { // Удаление элемента с верхушки стека.
    Item item = first.item;
    first = first.next;
    N--;
    return item;
  }
  // Реализацию iterator() см. в листинге 1.3.10.
  // Клиент тестирования main() приведен в листинге 1.3.7.
}
```

Эта обобщенная реализация API-интерфейса `Stack` основана на представлении связным списком. С ее помощью можно создавать стеки для любых типов данных. Для поддержки итерации добавьте выделенный код из листинга 1.3.10 для типа `Bag`.

```
% more tobe.txt
to be or not to - be - - that - - - is

% java Stack < tobe.txt
to be not that or be (в стеке осталось 2)
```

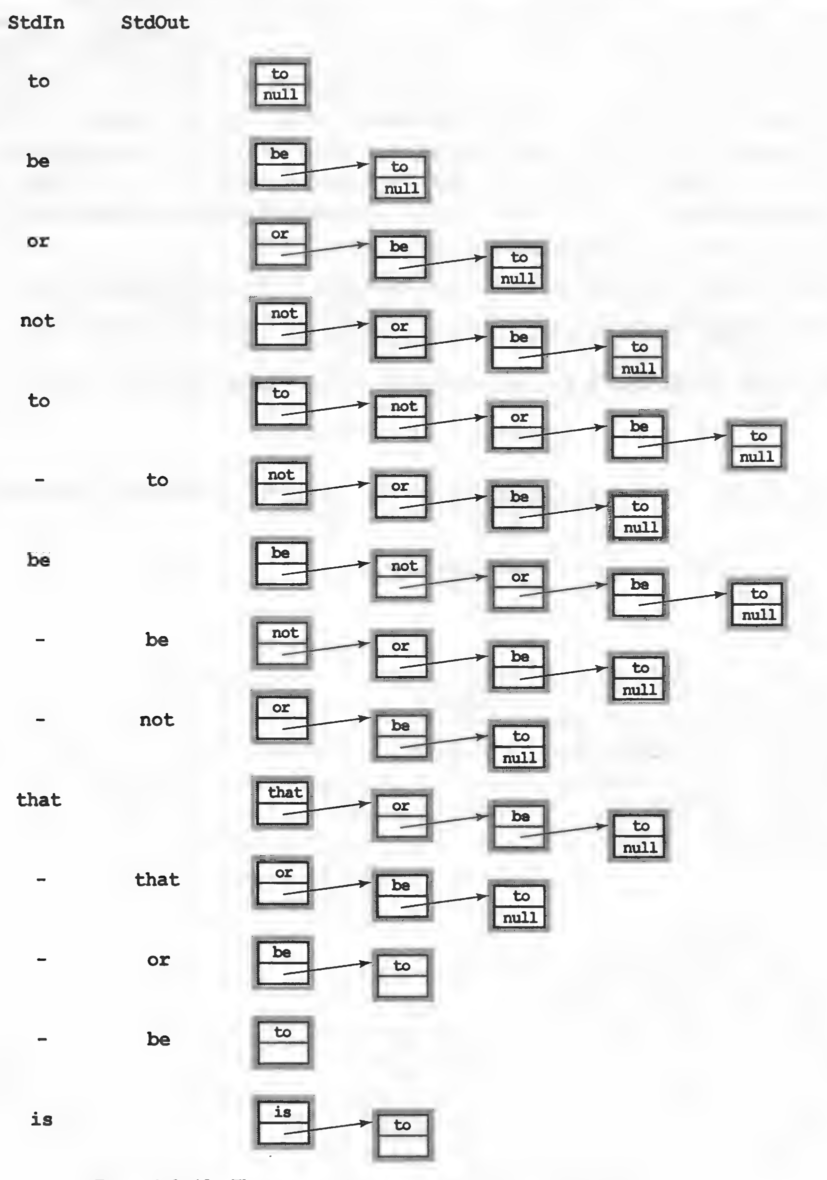


Рис. 1.3.12. Трассировка клиента тестирования `Stack`

Листинг 1.3.7. Клиент тестирования для класса Stack

```
public static void main(String[] args)
{  // Создание стека и вталкивание/выталкивание строк
   // в соответствии с командами из StdIn.
   Stack<String> s = new Stack<String>();
   while (!StdIn.isEmpty())
   {
      String item = StdIn.readString();
      if (!item.equals("-"))
          s.push(item);
      else if (!s.isEmpty()) StdOut.print(s.pop() + " ");
   }
   StdOut.println("(в стеке осталось " + s.size() + ")");
}
```

Эта реализация представляет собой прототип для многих рассматриваемых реализаций *алгоритмов*. В ней определена *структура данных* связного списка и реализованы методы push() и pop(), которые выполняют нужные действия с помощью лишь нескольких строк кода. Алгоритмы и структуры данных всегда идут бок о бок. В данном случае код реализации алгоритма довольно прост, но свойства структуры данных совсем не тривиальны и потребовали пояснений на нескольких предыдущих страницах. Подобная взаимосвязь определения структуры данных и реализации алгоритма типична и лежит в основе наших реализаций АТД в настоящей книге.

Реализация очереди

Реализация нашего API-интерфейса Queue на основе структуры данных связного списка также имеет простой вид — см. алгоритм 1.3 в листинге 1.3.8. Для представления очереди в ней используется связный список, содержащий элементы в порядке от самых "старых" до самых "новых" элементов. На начало очереди указывает переменная экземпляров first, а на конец — переменная last. Чтобы занести в очередь новый элемент, он добавляется в конец списка (как на рис. 1.3.11, только если список пуст, в ссылки first и last заносится адрес нового узла), а чтобы извлечь элемент, он удаляется из начала списка (как в методе pop() из класса Stack, но с изменением last, если список опустошается). Реализации методов size() и isEmpty() такие же, как в классе Stack. Как и в реализации Stack, здесь используется параметр обобщенного типа Item, и опущен код поддержки итерации, который будет приведен в реализации класса Bag (листинг 1.3.10).

Клиент тестирования, похожий на клиент для класса Stack, показан в листинге 1.3.9, а трассировка его работы — на рис. 1.3.13. В этой реализации используется такая же *структура данных*, как и для класса Stack, но в ней задействованы другие *алгоритмы* для добавления и удаления элементов, которые и отличают правило LIFO от FIFO. Здесь использование связных списков также обеспечивает оптимальную производительность: реализация применима для любых типов данных, объем необходимой памяти пропорционален количеству элементов в коллекции, а время, требуемое для выполнения любой операции, не зависит от размера коллекции.

Листинг 1.3.8. Алгоритм 1.3. Очередь

```
public class Queue<Item> implements Iterable<Item>
{
   private Node first;          // ссылка на самый "старый" узел
   private Node last;           // ссылка на самый "свежий" узел
   private int N;               // количество элементов в очереди

   private  class Node
   {  // вложенный класс для определения узлов
      Item item;
      Node next;
   }

   public boolean isEmpty() { return first == null; }  // Или N == 0.
   public int size()        { return N;  }

   public void enqueue(Item item)
   {  // Добавление элемента в конец списка.
      Node oldlast = last;
      last = new Node();
      last.item = item;
      last.next = null;
      if (isEmpty()) first = last;
      else           oldlast.next = last;
      N++;
   }

   public Item dequeue()
   {  // Удаление элемента из начала списка.
      Item item = first.item;
      first = first.next;
      if (isEmpty()) last = null;
      N--;
      return item;
   }

   // Реализацию iterator() см. в листинге 1.3.10.
   // Клиент тестирования main() приведен в листинге 1.3.9.
}
```

Эта обобщенная реализация API-интерфейса Queue основана на представлении связным списком. С ее помощью можно создавать очереди для любых типов данных. Для поддержки итерации добавьте выделенный код из листинга 1.3.10 для типа Bag.

```
% more tobe.txt
to be or not to - be - - that - - - is

% java Queue < tobe.txt
to be not that or be (в очереди осталось 2)
```

Листинг 1.3.9. Клиент тестирования для класса Queue

```
public static void main(String[] args)
{  // Создание очереди и занесение/извлечение строк.

   Queue<String> q = new Queue<String>();
```

```
    while (!StdIn.isEmpty())
    {
      String item = StdIn.readString();
      if (!item.equals("-"))
          q.enqueue(item);
      else if (!q.isEmpty()) StdOut.print(q.dequeue() + " ");
    }
    StdOut.println("(в очереди осталось " + q.size() + ")");
}
```

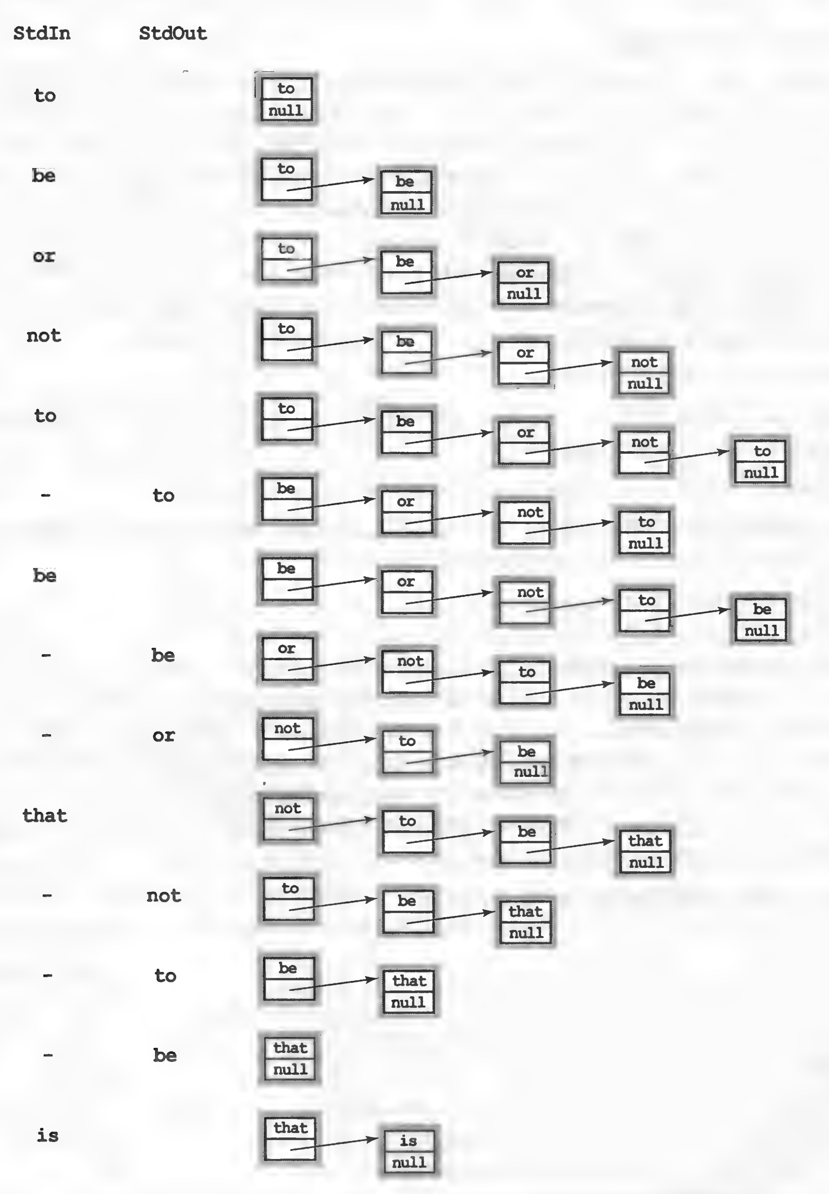


Рис. 1.3.13. Трассировка клиента тестирования Queue

Связные списки — фундаментальная альтернатива массивам для структурирования коллекций данных. Эта альтернатива доступна программистам уже много десятилетий. Заметной вехой в истории языков программирования была разработка языка LISP Джоном Маккарти в 1950-х годах — и в нем связные списки являются основными структурами для программ и данных. Программирование, в котором задействованы связные списки, сопряжено с различными трудностями — в том числе и при отладке, как будет показано в упражнениях. В настоящее время использование безопасных указателей, автоматическая сборка мусора и абстрактные типы данных позволяют инкапсулировать код обработки списков в лишь нескольких классах, вроде приведенных здесь.

Реализация контейнера

Для реализации нашего API-интерфейса `Bag` с помощью структуры данных связного списка достаточно просто заменить в классе `Stack` имя метода `push()` на `add()` и удалить метод `pop()` — получится алгоритм 1.4 (см. листинг 1.3.10). То же самое можно получить и из класса `Queue`, но изменений потребуется немного больше. В этой реализации выделен код, необходимый для выполнения итерации в типах `Stack`, `Queue` и `Bag` — т.е. для прохода по списку. В классе `Stack` проход выполняется по правилу LIFO, в классе `Queue` — по правилу FIFO, а в классе `Bag` — тоже по правилу LIFO, но порядок прохода там не важен. Как видно из выделенного кода алгоритма 1.4, для реализации итерации в коллекции вначале нужно включить библиотеку:

```
import java.util.Iterator;
```

чтобы код мог обращаться к Java-интерфейсу `Iterator`. Кроме того, в объявление класса нужно добавить конструкцию

```
implements  Iterable<Item>
```

как обещание предоставить метод `iterator()`. Данный метод просто возвращает объект из класса, который реализует интерфейс `Iterator`:

```
public Iterator<Item> iterator()
{ return new ListIterator(); }
```

Этот код обещает реализовать класс, который содержит методы `hasNext()`, `next()` и `remove()`, вызываемые при использовании конструкции *foreach*. Для реализации этих методов вложенный класс `ListIterator` из алгоритма 1.4 использует переменную экземпляров `current`, которая отслеживает текущий узел при проходе по списку. Затем метод `hasNext()` проверяет значение `current` на равенство `null`, и метод `next()` сохраняет ссылку на текущий элемент, перезаписывает в `current` ссылку на следующий узел списка и возвращает сохраненную ссылку.

Листинг 1.3.10. Алгоритм 1.4. Контейнер

```
import java.util.Iterator;
public class  Bag<Item> implements Iterable<Item>
{
   private Node first;  // первый узел списка
   private  class Node
   {
      Item item;
      Node next;
   }
   public void add(Item item)
```

```
   { // совпадает с push() из Stack
      Node oldfirst = first;
      first = new Node();
      first.item = item;
      first.next = oldfirst;
   }
   public Iterator<Item> iterator()
   { return new ListIterator(); }
   private class ListIterator implements Iterator<Item>
   {
      private Node current = first;
      public boolean hasNext()
      { return current != null; }
      public void remove() { }
      public Item next()
      {
         Item item = current.item;
         current = current.next;
         return item;
      }
   }
}
```

Эта реализация API-интерфейса `Bag` использует связный список элементов, добавляемых вызовами `add()`. Код методов `isEmpty()` и `size()` совпадает с кодом из класса `Stack` и поэтому опущен. Итератор выполняет проход по списку, сохраняя ссылку на текущий узел в переменной `current`. Классы `Stack` и `Queue` можно сделать итерируемыми, добавив выделенный код в алгоритмы 1.1 и 1.2, поскольку они используют ту же базовую структуру данных, только обеспечивают в них выполнение правила LIFO или FIFO соответственно.

Обзор

Приведенные в данном разделе реализации контейнеров, очередей и стеков, с поддержкой обобщений и итерации, предоставляют нам уровень абстракции, который позволяет писать компактные клиентские программы для обработки коллекций объектов. Четкое понимание этих АТД важно в качестве введения в изучение алгоритмов и структур данных по трем причинам. Во-первых, мы используем эти типы данных в качестве строительных блоков в структурах данных более высокого уровня. Во-вторых, они демонстрируют взаимосвязь структур данных и алгоритмов, а также возможные трудности в достижении различных показателей эффективности. В-третьих, назначение нескольких наших реализаций состоит в том, чтобы познакомиться с АТД, которые поддерживают более мощные операции с коллекциями объектов, и мы будем использовать приведенные здесь реализации в качестве отправной точки.

Структуры данных

Теперь мы можем представлять коллекции объектов двумя способами: в виде массивов и связных списков. Массивы встроены в язык Java, а связные списки легко создавать с помощью стандартных записей (табл. 1.3.4). Эти две фундаментальные альтернативы часто называют *последовательным расположением* и *связным расположением*. Ниже в данной книге мы разработаем реализации АТД, в которых эти базовые структуры будут различным образом сочетаться и расширяться. Одно важное расширение — структуры данных с несколькими ссылками. Например, в разделах 3.2 и 3.3 мы будем рассматривать структу-

ры данных, которые называются *бинарными деревьями* и состоят из узлов с *двумя* ссылками каждый. Еще одно важное расширение — *составные* структуры данных: можно иметь контейнер стеков, очередь массивов и т.д. Например, в главе 4 мы будем рассматривать графы, представленные в виде массивов контейнеров. Подобным образом нетрудно определять структуры данных произвольной сложности, и одной важной причиной нашего изучения абстрактных типов данных является желание управлять этой сложностью.

Таблица 1.3.4. Фундаментальные структуры данных

Структура данных	Преимущество	Недостаток
Массив	К любому элементу можно непосредственно обратиться по индексу	Необходимо знать размер при инициализации
Связный список	Объем необходимой памяти пропорционален размеру	Для доступа к элементу необходима ссылка

Наше обращение с контейнерами, очередями и стеками в данном разделе представляет собой прототипный пример подхода, который будет использоваться во всей книге для описания структур данных и алгоритмов. При рассмотрении новых областей мы будем определять вычислительную сложность и использовать абстракции данных для их преодоления, выполняя перечисленные ниже шаги.

- Формулировка API-интерфейса.
- Разработка клиентского кода, ориентированного на конкретные области применения.
- Описание структуры данных (представления набора значений), пригодной в качестве основы для *переменных экземпляров* в классе, реализующем АТД, который удовлетворяет спецификациям из API-интерфейса (табл. 1.3.5).
- Описание алгоритмов (подходов к реализации множества операций), которые могут служить основой для реализации *методов экземпляров* в классе.
- Анализ характеристик производительности алгоритмов.

В следующем разделе мы вплотную займемся последним шагом, поскольку он часто определяет, какие алгоритмы и реализации могут оказаться наиболее удобными для решения реальных задач.

Таблица 1.3.5. Примеры структур данных, разработанных в данной книге

Структура данных	Раздел	АТД	Представление
Дерево с родительскими ссылками	1.5	`UnionFind`	Массив целых чисел
Дерево бинарного поиска	3.2, 3.3	`BST`	Две ссылки на узел
Строка	5.1	`String`	Массив, смещение и длина
Бинарное пирамидальное дерево	2.4	`PQ`	Массив объектов
Хеш-таблица (раздельные цепочки)	3.4	`SeparateChainingHashST`	Массив связных списков
Хеш-таблица (линейное опробование)	3.4	`LinearProbingHashST`	Два массива объектов
Списки смежности графа	4.1, 4.2	`Graph`	Массив объектов `Bag`
Trie-дерево	5.2	`TrieST`	Узел с массивом ссылок
Дерево тернарного поиска	5.3	`TST`	Три ссылки на узел

Вопросы и ответы

Вопрос. **Обобщения отсутствуют во многих языках программирования, в том числе и в ранних версиях Java. Какие есть альтернативы?**

Ответ. Одна из альтернатив — использование различных реализаций для каждого типа данных, как и было сказано в тексте. Еще одна — создание стека значений `Object` и приведение в методе `pop()` к типу, нужному в клиентском коде. Неудобство такого подхода в том, что до выполнения программы невозможно выявить ошибки несоответствия типа. А если использовать обобщения и написать код для вталкивания в стек объекта не того типа:

```
Stack<Apple> stack = new Stack<Apple>();
Apple  a = new Apple();
...
Orange b = new Orange();
...
stack.push(a);
...
stack.push(b);     // ошибка на этапе компиляции
```

то возникнет ошибка времени компиляции:

```
push(Apple) in Stack<Apple> cannot be applied to (Orange)
```

Даже только одна эта возможность обнаруживать такие ошибки достаточна для использования обобщений.

Вопрос. **Почему в Java недопустимы обобщенные массивы?**

Ответ. Эксперты все еще не пришли к единому мнению по этому вопросу. Чтобы разобраться в нем, вам надо стать одним из них! А новичкам будет полезно узнать о *ковариантных массивах* и *разрушении типа*.

Вопрос. **Как создать массив стеков строк?**

Ответ. Используйте приведение такого вида:

```
Stack<String>[] a = (Stack<String>[]) new Stack[N];
```

Внимание! Подобное приведение в клиентском коде отличается от описанного в подразделе "Обобщения". Возможно, вы думали, что следует использовать `Object`, а не **`Stack`**. При использовании обобщений Java проверяет совместимость типов на этапе компиляции, но не учитывает эту информацию во время выполнения, поэтому остается `Stack<Object>[]` или, короче, просто `Stack[]`, и необходимо привести этот тип к `Stack<String>[]`.

Вопрос. **Что произойдет, если программа вызовет `pop()` для пустого стека?**

Ответ. Это зависит от реализации. В нашей книжной реализации (алгоритм 1.2) возникнет исключение `NullPointerException`. В реализации на сайте книги генерируется исключение времени выполнения, которое поможет пользователям локализовать ошибку. В общем случае рекомендуется вставлять как можно больше проверок в код, который будет использоваться многими людьми.

Вопрос. **Зачем было возиться с изменением размера массивов, если есть связные списки?**

Ответ. Мы познакомимся с несколькими примерами реализации АТД, где для выполнения некоторых операций удобнее использовать массивы и гораздо сложнее — связные списки. Класс `ResizingArrayStack` представляет собой модель для управления расходом памяти в таких АТД.

Вопрос. **Зачем объявлять Node как вложенный класс? И почему он private?**

Ответ. С помощью объявления вложенного класса Node как private доступ к его методам и переменным экземпляров ограничивается пределами объемлющего класса и невозможен откуда-либо еще, поэтому нет необходимости описывать переменные экземпляров как public или private. *Примечание для экспертов*: не статический вложенный класс называется *внутренним* классом, поэтому формально наши классы Node являются внутренними, хотя не обобщенные классы могут быть статическими.

Вопрос. **При вводе команды javac Stack.java для выполнения алгоритма 1.2 и аналогичных программ обнаруживается файл Stack.class и файл Stack$Node.class. Для чего нужен второй из них?**

Ответ. Это файл для внутреннего класса Node. По соглашениям об именовании в Java символ $ применяется для отделения имени внешнего класса от внутреннего.

Вопрос. **Имеются ли в Java библиотеки для стеков и очередей?**

Ответ. И да, и нет. В Java имеется встроенная библиотека java.util.Stack, но если вам нужен именно стек, то лучше ею не пользоваться. В этой библиотеке имеются несколько дополнительных операций, не характерных для стека — например, выборка i-го элемента. Кроме того, в нем можно добавить элемент на дно стека (а не на верхушку), т.е. он может имитировать очередь! Эти дополнительные операции могут иногда помочь, но обычно они мешают. Мы используем типы данных не как библиотеки всех операций, которые только можно выдумать, а в первую очередь как механизм для точного указания нужных операций. Основное преимущество этого подхода в том, что система может помешать в выполнении операций, которые реально нам не нужны. API-интерфейс java.util.Stack является примером *широкого интерфейса*, который мы будем стараться обходить стороной.

Вопрос. **Следует ли разрешить клиенту вставлять в стек или очередь нулевые (null) элементы?**

Ответ. Этот вопрос часто возникает при реализациях коллекций в Java. В наших реализациях (и библиотечных Java-реализациях стека и очереди) вставка значений null разрешена.

Вопрос. **Как должен среагировать итератор Stack, если во время его работы клиент вызовет метод push() или pop()?**

Ответ. Сгенерируйте исключение java.util.ConcurrentModificationException, чтобы обеспечить для итератора *быстрый сбой*. См. упражнение 1.3.50.

Вопрос. **Можно ли использовать цикл *foreach* с массивами?**

Ответ. Да (хотя массивы и не реализуют интерфейс Iterable). Следующий короткий фрагмент выводит аргументы командной строки:

```
public static void main(String[] args)
{ for (String s : args) StdOut.println(s); }
```

Вопрос. **Можно ли использовать цикл *foreach* со строками?**

Ответ. Нет. Класс String не реализует интерфейс Iterable.

Вопрос. **Почему бы не иметь один тип данных `Collection`, реализующий методы добавления элементов, удаления самого свежего, удаления самого старого, удаление случайного элемента, итерации, возврата количества элементов в коллекции и других нужных нам операций? Тогда они были бы все упакованы в один класс, который может использоваться многими клиентами.**

Ответ. Это еще один пример *широкого интерфейса*. В Java есть такие реализации — классы `java.util.ArrayList` и `java.util.LinkedList`. Их лучше избегать, в частности, потому, что нет гарантии, что все операции реализованы эффективно. В этой книге мы используем в качестве отправных точек для проектирования эффективных алгоритмов и структур данных такие API-интерфейсы, которые содержат лишь несколько операций. Еще одна причина придерживаться узких интерфейсов — они несколько дисциплинируют клиентские программы, в результате чего их код гораздо легче понимать. Если в одном клиенте используется `Stack<String>`, а в другом `Queue<Transaction>`, то несложно догадаться, что в первом случае важно правило LIFO, а во втором — правило FIFO.

Упражнения

1.3.1. Добавьте в класс `FixedCapacityStackOfStrings` метод `isFull()`.

1.3.2. Что выведет команда `java Stack` для следующих входных данных?

```
it was - the best - of times - - - it was - the - -
```

1.3.3. Предположим, что клиент выполняет смесь (стековых) операций *втолкнуть* и *вытолкнуть*. Операции вталкивания помещают на верхушку стека целые числа от 0 до 9 по порядку; а операции выталкивания выводят удаляемые значения. Какие из следующих последовательностей *не* могут появиться в качестве выходных?

а) 4 3 2 1 0 9 8 7 6 5
б) 4 6 8 7 5 3 2 9 0 1
в) 2 5 6 7 4 8 9 3 1 0
г) 4 3 2 1 0 5 6 7 8 9
д) 1 2 3 4 5 6 9 8 7 0
е) 0 4 6 5 3 8 1 7 2 9
ж) 1 4 7 9 8 6 5 3 0 2
з) 2 1 4 3 6 5 8 7 9 0

1.3.4. Напишите клиент стека `Parentheses`, который читает поток символов из стандартного ввода и использует стек для определения правильности балансировки скобок. Например, программа должна вывести `true` для `[()]{}{[()()]()}` и `false` для `[(])`.

1.3.5. Что выведет следующий кодовый фрагмент, если `N` равно 50? Приведите высокоуровневое описание его действия для положительного целого числа `N` на входе.

```
Stack<Integer> stack = new Stack<Integer>();
while (N > 0)
{
   stack.push(N % 2);
   N = N / 2;
}
for (int d : stack) StdOut.print(d);
StdOut.println();
```

Ответ: этот код выводит двоичное представление `N` (`110010`, если `N` равно 50).

1.3.6. Что делает следующий кодовый фрагмент с очередью q?

```
Stack<String> stack = new Stack<String>();
while (!q.isEmpty())
   stack.push(q.dequeue());
while (!stack.isEmpty())
   q.enqueue(stack.pop());
```

1.3.7. Добавьте в класс Stack метод peek(), который возвращает элемент, занесенный в стек последним (без выталкивания).

1.3.8. Приведите содержимое и размер массива из класса DoublingStackOfStrings после обработки входных данных

```
it was - the best - of times - - - it was - the - -
```

1.3.9. Напишите программу, которая принимает из стандартного ввода выражение без левых скобок и выводит эквивалентное инфиксное выражение со вставленными недостающими скобками. Например, для входных данных

```
1 + 2 ) * 3 - 4 ) * 5 - 6 ) ) )
```

программа должна вывести

```
( ( 1 + 2 ) * ( ( 3 - 4 ) * ( 5 - 6 ) ) )
```

1.3.10. Напишите фильтр InfixToPostfix, преобразующий арифметическое выражение из инфиксного вида в постфиксный.

1.3.11. Напишите программу EvaluatePostfix, которая принимает из стандартного ввода постфиксное выражение, вычисляет его и выводит результат. (Передача по конвейеру выходных данных программы из предыдущего упражнения на вход этой программы эквивалентно поведению класса Evaluate.)

1.3.12. Напишите *клиент* класса Stack с возможностью итерации и статическим методом copy(), который принимает в качестве аргумента стек строк и возвращает копию этого стека. *Примечание*: эта возможность — наглядный пример пользы итератора, поскольку он позволяет разрабатывать такие методы без изменения базового API-интерфейса.

1.3.13. Предположим, что клиент выполняет смешанную последовательность операций *занести* и *извлечь* (для очереди). Операции занесения добавляют в очередь целые числа от 0 до 9 по порядку, а операции извлечения выводят удаляемые значения. Какие из следующих последовательностей не могут появиться в качестве выходных?

а) 0 1 2 3 4 5 6 7 8 9

б) 4 6 8 7 5 3 2 9 0 1

в) 2 5 6 7 4 8 9 3 1 0

г) 4 3 2 1 0 5 6 7 8 9

1.3.14. Разработайте класс ResizingArrayQueueOfStrings, который реализует абстракцию очереди с массивом фиксированного размера, а потом добавьте в полученную реализацию изменение размера массива, чтобы снять ограничение на размер.

1.3.15. Напишите клиент класса Queue, который принимает в командной строке аргумент k и выводит k-тую с конца строку из введенных из стандартного ввода (предполагается, что стандартный ввод содержит k или более строк).

1.3.16. Используйте в качестве образца метод `readInts()` из листинга 1.3.2 и напишите статистический метод `readDates()` для класса `Date`, который читает из стандартного ввода даты в формате, заданном в табл. 1.2.7, и возвращает массив, содержащий эти даты.

1.3.17. Выполните упражнение 1.3.16 для класса `Transaction`.

Упражнения со связными списками

Эти упражнения помогут вам обрести опыт работы со связными списками. Рекомендуем использовать чертежи вроде приведенных в тексте данного раздела.

1.3.18. Пусть `x` — не последний узел связного списка. Что делает следующий кодовый фрагмент?

```
x.next = x.next.next;
```

Ответ: удаляет из списка узел, непосредственно следующий за `x`.

1.3.19. Приведите кодовый фрагмент, удаляющий последний узел из связного списка, на первый узел которого указывает ссылка `first`.

1.3.20. Напишите метод `delete()`, который принимает целочисленный аргумент `k` и удаляет из связного списка `k`-тый элемент, если он существует.

1.3.21. Напишите метод `find()`, который принимает в качестве аргументов связный список и строку `key` и возвращает `true`, если какой-то узел списка содержит в поле элемента значение `key`, и `false` в противном случае.

1.3.22. Пусть `x` — узел связного списка. Что делает следующий кодовый фрагмент?

```
t.next = x.next;
x.next = t;
```

Ответ: вставляет узел `t` сразу за узлом `x`.

1.3.23. Почему следующий кодовый фрагмент не делает то же самое, что и код из предыдущего упражнения?

```
x.next = t;
t.next = x.next;
```

Ответ: перед обновлением ссылки `t.next` значение `x.next` указывает уже не на узел, который первоначально следовал за `x`, а на сам узел `t`!

1.3.24. Напишите метод `removeAfter()`, который принимает в качестве аргумента узел связного списка и удаляет узел, следующий за ним (и не делает ничего, если в поле `next` узла-аргумента находится нулевая ссылка).

1.3.25. Напишите метод `insertAfter()`, который принимает в качестве аргументов два узла связного списка и вставляет второй из них в список после первого (и не делает ничего, если любой из аргументов равен `null`).

1.3.26. Напишите метод `remove()`, который принимает в качестве аргументов связный список и строку `key` и удаляет из списка все узлы, содержащие в поле элемента значение `key`.

1.3.27. Напишите метод `max()`, который принимает в качестве аргумента первый узел связного списка и возвращает значение максимального элемента в этом списке. Все элементы представляют собой положительные целые числа. В случае пустого списка нужно возвратить значение `0`.

1.3.28. Разработайте рекурсивное решение предыдущего упражнения.

1.3.29. Напишите реализацию класса `Queue`, в которой используется *кольцевой* связный список — такой же, как и обычный связный список, но без нулевых ссылок, т.е. `last.next` содержит значение `first`, если список не пуст. Используйте только одну переменную экземпляров `Node` (`last`).

1.3.30. Напишите функцию, которая принимает в качестве аргумента первый узел связного списка, обращает порядок элементов списка (с разрушением исходного) и возвращает в качестве результата первый узел полученного списка.

Итеративное решение. Для решения этой задачи нам потребуются ссылки на три узла в связных списках: `reverse`, `first` и `second`. На каждой итерации извлекается узел `first` из исходного связного списка и вставляется в начало обращенного списка. В любой момент верно, что `first` — первый узел (остатка) первого списка, `second` — второй узел (остатка) первого списка, а `reverse` — первый узел результирующего обращенного списка.

```
public Node reverse(Node x)
{
   Node first   = x;
   Node reverse = null;
   while (first != null)
   {
      Node second = first.next;
      first.next  = reverse;
      reverse     = first;
      first       = second;
   }
   return reverse;
}
```

При написании кода, обрабатывающего связные списки, всегда следует обращать пристальное внимание на нестандартные ситуации (когда список пуст, когда список содержит только один или два узла) и граничные случаи (обработка первого и последнего элементов). Такие ситуации существенно усложняют программу.

Рекурсивное решение. Пусть связный список содержит N узлов, тогда нужно рекурсивно обратить порядок последних $N-1$ узлов, а затем добавить в конец первый узел.

```
public Node reverse(Node first)
{
   if (first == null) return null;
   if (first.next == null) return first;
   Node second = first.next;
   Node rest = reverse(second);
   second.next = first;
   first.next  = null;
   return rest;
}
```

1.3.31. Реализуйте вложенный класс `DoubleNode` для построения двухсвязных списков, в которых каждый узел содержит ссылку на узел, предшествующий ему в списке, и ссылку на следующий узел (`null`, если такого элемента нет). Затем реализуйте статические методы для решения следующих задач: вставка в начало, вставка в конец, удаление из начала, удаление из конца, вставка перед указным узлом, вставка после указанного узла и удаление указанного узла.

Творческие задачи

1.3.32. *Стеко-очередь.* Очередь с элементами поведения стека, или *стеко-очередь* — это тип данных, поддерживающий операции *втолкнуть*, *вытолкнуть* и *занести*. Сформулируйте API-интерфейс для такого АТД. Разработайте реализацию на основе связного списка.

1.3.33. *Дек.* Очередь с двумя концами, или *дек* (double-ended queue — deque), похожа на стек или очередь, но поддерживает добавление и удаление элементов с обоих концов. Дек хранит коллекцию элементов и поддерживает следующий API-интерфейс:

public class **Deque**<Item> implements Iterable<Item>

	Deque()	*создание пустого дека*
boolean	isEmpty()	*пуст ли дек?*
int	size()	*количество элементов в деке*
void	pushLeft(Item item)	*добавление элемента с левого конца*
void	pushRight(Item item)	*добавление элемента с правого конца*
Item	popLeft()	*удаление элемента с левого конца*
Item	popRight()	*удаление элемента с правого конца*

Напишите класс Deque, который реализует этот API-интерфейс с помощью двухсвязного списка, и класс ResizingArrayDeque, который использует массив с переменным размером.

1.3.34. *Случайный контейнер. Случайный контейнер* содержит коллекцию элементов и поддерживает следующий API-интерфейс:

public class **RandomBag**<Item> implements Iterable<Item>

	RandomBag()	*создание пустого случайного контейнера*
boolean	isEmpty()	*пуст ли контейнер?*
int	size()	*количество элементов в контейнере*
void	add(Item item)	*добавление элемента*

Напишите класс RandomBag, который реализует этот API-интерфейс. Обратите внимание, что он практически совпадает с классом Bag, кроме определения *случайный* — оно означает, что итерация должна выдавать элементы в *случайном* порядке (все $N!$ перестановок равновероятны для каждого итератора). *Совет*: в конструкторе итератора поместите элементы в массив и перемешайте их.

1.3.35. *Случайная очередь. Случайная очередь* содержит коллекцию элементов и поддерживает следующий API-интерфейс:

public class **RandomQueue**<Item>

	RandomQueue()	*создание пустой случайной очереди*
boolean	isEmpty()	*пуста ли очередь?*
void	enqueue(Item item)	*добавление элемента*
Item	dequeue()	*извлечение и возврат случайного элемента (выборка без замены)*
Item	sample()	*возврат случайного элемента без удаления (выборка с заменой)*

6 Зак. 3239

Напишите класс RandomBag, который реализует этот API-интерфейс. *Совет*: используйте представление посредством массива (с подгонкой размера). Для удаления элемента обменяйте элемент в случайной позиции (от 0 до N-1) с элементом в последней позиции (индекс N-1). Затем удалите и возвратите последний объект, как в классе ResizingArrayStack. Напишите клиент, который раздает карты для игры в бридж (по 13 карт) с помощью конкретизации RandomQueue<Card>.

1.3.36. *Случайный итератор*. Напишите итератор для типа RandomQueue<Item> из предыдущего упражнения, который возвращает элементы в случайном порядке.

1.3.37. *Задача Иосифа*. Эта задача известна из глубокой древности. N человекам нужно было выбрать одного, и для этого они встали в круг (позиции с номерами от 0 до $N-1$) и считали по кругу, удаляя каждого M-го человека, пока не остался один. Легенда гласит, что Иосиф Флавий вычислил то место, которое остается последним. Напишите клиент Josephus для класса Queue, который принимает из командной строки числа N и M и выводит порядок, в котором выбывают люди (и таким образом подсказывает Иосифу, какое место в круге следует выбрать).

```
% java Josephus 7 2
1 3 5 0 4 2 6
```

1.3.38. *Удаление* k*-го элемента*. Реализуйте класс, который поддерживает следующий API-интерфейс:

public class **GeneralizedQueue**<Item>

	GeneralizedQueue()	*создание пустой очереди*
boolean	isEmpty()	*пуста ли очередь?*
void	insert(Item x)	*добавление элемента*
Item	delete(int k)	*извлечение и возврат* k*-го из последних вставленных элементов*

Вначале разработайте реализацию с использованием массива, а затем с использованием связного списка. *Примечание*: алгоритмы и структуры данных, рассматриваемые в главе 3, позволяют разработать реализацию со временем выполнения методов insert() и delete(), гарантированно пропорциональным логарифму количества элементов в очереди — см. упражнение 3.5.27.

1.3.39. *Кольцевой буфер*. Кольцевой буфер, или кольцевая очередь — это структура данных с правилом FIFO фиксированного размера N, удобная для передачи данных между асинхронными процессами или для хранения файлов журналов. Если буфер пуст, получатель ждет поступления в него данных; если буфер полон, отправитель ждет, когда можно будет поместить данные. Разработайте API-интерфейс для типа RingBuffer и реализацию на основе массива (с закольцовкой).

1.3.40. *Сдвиг в начало*. Напишите программу для чтения символов из стандартного ввода и сохранения их в связном списке без повторов. При вводе символа, который раньше не встречался, вставьте его в начало списка. При вводе уже знакомого символа удалите его из списка и снова вставьте в начало. Назовите программу MoveToFront: она реализует известную стратегию *сдвига в начало* (move to front), которая удобна при кешировании, сжатии данных и во многих других приложениях, где недавно обработанные элементы с большей вероятностью могут встретиться вновь.

1.3.41. *Копирование очереди.* Напишите новый конструктор — такой, что после выполнения оператора

```
Queue<Item> r = new Queue<Item>(q);
```

ссылка `r` будет указывать на новую независимую копию очереди `q`. Занесение и извлечение элементов в одной очереди не должно как-то влиять на другую очередь. *Совет*: удалите все элементы из `q`, добавляя их одновременно в `q` и `r`.

1.3.42. *Копирование стека.* Напишите новый конструктор для реализации класса `Stack` на основе связного списка — такой, что оператор

```
Stack<Item> t = new Stack<Item>(s);
```

создает ссылку `t` на новую независимую копию стека `s`.

1.3.43. *Список файлов.* Папка — это список файлов и папок. Напишите программу, которая принимает из командной строки в качестве аргумента имя папки и выводит все файлы, содержащиеся в этой папке, причем содержимое каждой папки рекурсивно выводится (со сдвигом вправо) под именем этой папки. *Совет*: используйте очередь и библиотеку `java.io.File`.

1.3.44. *Буфер текстового редактора.* Разработайте тип данных для буфера текстового редактора, который реализует следующий API-интерфейс:

`public class` **`Buffer`**

	`Buffer()`	*создание пустого буфера*
`void`	`insert(char c)`	*вставка символа* `c` *в позиции курсора*
`char`	`delete()`	*удаление и возврат символа в позиции курсора*
`void`	`left(int k)`	*сдвиг курсора на* `k` *позиций влево*
`void`	`right(int k)`	*сдвиг курсора на* `k` *позиций вправо*
`int`	`size()`	*количество символов в буфере*

Совет: используйте два стека.

1.3.45. *Применимость стека.* Пусть имеется смесь операций *втолкнуть* и *вытолкнуть*, как в случае нашего клиента тестирования для стека, где целые числа $0, 1, ..., N-1$ по порядку (директивы *втолкнуть*) перемешаны с N дефисами (директивы *вытолкнуть*). Разработайте алгоритм, который определяет, приведет ли данная смесь директив к истощению стека. Можно использовать лишь объем памяти, не зависящий от N, т.е. нельзя хранить числа в какой-то структуре данных. Разработайте также алгоритм, который определяет, может ли заданная перестановка целых чисел появиться в качестве выходных данных нашего клиента тестирования (в зависимости от того, где находятся директивы *вытолкнуть*).

Решение. Стек может истощиться только в том случае, если существует такое целое число k, что первые k операций выталкивания заданы раньше первых k операций вталкивания. Если некоторая перестановка может быть сгенерирована, то она генерируется уникальным образом: если очередное целое число из перестановки находится на верхушке стека, нужно вытолкнуть его, иначе нужно втолкнуть его в стек.

1.3.46. *Применимость стека с запрещенными тройками.* Докажите, что перестановка может быть сгенерирована стеком (по принципу, описанному в предыдущем упражнении) тогда и только тогда, когда в ней нет запрещенных троек чисел (a, b, c) таких, что $a < b < c$, и c идет в перестановке первым, a вторым, а b — третьим (возможно, с другими числами между ними).

Частичное решение. Предположим, что в перестановке имеется запрещенная тройка (a, b, c). Элемент c вытолкнут до a и b, но a и b втолкнуты раньше c — т.е. на момент вталкивания c и a, и b уже находятся в стеке. Следовательно, a не может быть вытолкнут прежде b.

1.3.47. *Конкатенируемые очереди, стеки или стеко-очереди.* Разработайте дополнительную операцию *катенация*, которая конкатенирует (с разрушением) две очереди, стека или стеко-очереди (см. упражнение 1.3.32). *Совет*: используйте кольцевой связный список с указателем на последний элемент.

1.3.48. *Два стека в деке.* Реализуйте два стека в одном деке, так, чтобы для выполнения каждой операции понадобилось постоянное количество операций с деком (см. упражнение 1.3.33).

1.3.49. *Очередь на основе стеков.* Реализуйте очередь с помощью фиксированного количества стеков, чтобы каждая операция с очередью требовала выполнения постоянного (в худшем случае) количества операций со стеками. *Внимание*: задача очень сложная!

1.3.50. *Итератор с быстрым сбоем.* Измените код итератора для класса `Stack` так, чтобы он немедленно генерировал исключение `java.util.ConcurrentModificationException`, если в процессе итерации клиент изменяет коллекцию с помощью операций `push()` или `pop()`.

Решение. Используйте счетчик, который подсчитывает количество операций `push()` и `pop()`. При создании итератора сохраните это значение в переменной экземпляра `Iterator`. Перед каждым вызовом `hasNext()` и `next()` проверяйте, изменилось ли это количество с момента создания итератора, и если изменилось, генерируйте исключение.

1.4. Анализ алгоритмов

По мере освоения компьютеров люди стали использовать их для решения все более сложных задач и для обработки все больших объемов данных. При этом неизменно возникают примерно такие вопросы:

Сколько времени потребует программа?

Почему программе не хватает памяти?

Вы наверняка задавали себе подобные вопросы — при пересоздании библиотеки музыкальных или графических файлов, при установке нового приложения, при работе с крупным документом или при обработке большого количества экспериментальных данных. Но эти вопросы слишком расплывчаты, и ответы зависят от многих факторов: от свойств конкретного компьютера, от конкретных обрабатываемых данных и от конкретной программы, которая выполняет данную задачу (и реализует некоторый алгоритм). Все эти факторы приводят к анализу угнетающе большого объема информации.

Однако, несмотря на эти сложности, путь к получению полезных ответов на такие базовые вопросы часто весьма прост, что и будет показано в данном разделе. Этот процесс основан на *научном методе* — широко распространенном наборе приемов, используемых учеными для получения знаний об окружающем нас мире. Мы применяем *математический анализ* для разработки четких моделей трудоемкости и *экспериментальные исследования* для проверки этих моделей.

Научный метод

Тот же подход, который используют ученые для изучения реальных явлений, удобен и для изучения времени выполнения программ:

- *наблюдение* каких-то характеристик из реального мира, обычно на основе точных измерений;
- *предложение* гипотетической модели, которая согласуется с наблюдениями;
- *предсказание* событий на основе предложенной модели;
- *проверка* предсказаний с помощью дальнейших наблюдений;
- *обоснование* с помощью повторения процесса, пока гипотеза и наблюдения не совпадут.

Один из основных принципов научного подхода — разрабатываемые нами эксперименты должны быть *воспроизводимыми*, чтобы другие могли убедиться в правильности модели, самостоятельно проверив гипотезу. Гипотезы должны быть также *фальсифицируемыми*, чтобы можно было точно знать, когда гипотеза не верна (и, значит, требует пересмотра). Согласно знаменитому высказыванию, приписываемому Эйнштейну — "Никакой объем экспериментальных наблюдений не может доказать, что я прав, но один эксперимент может доказать, что я ошибаюсь" — мы никогда не можем быть абсолютно уверены, что гипотеза верна; мы можем лишь удостовериться, что она не противоречит нашим наблюдениям.

Наблюдения

Первый вопрос, на который нам необходимо найти ответ — как выполнить численные измерения времени выполнения программ? Ответ на него проще, чем в естественных науках. Нам не нужно посылать ракету на Марс, препарировать животных в лаборатории или расщеплять атом: достаточно просто выполнить программу. Вообще говоря, при *каждом* запуске программы мы выполняем научный эксперимент, который связывает программу с реальным миром и отвечает на один из наиважнейших вопросов: *насколько долго будет выполняться программа*?

Наше первое качественное наблюдение касается большинства программ: существует *размер задачи*, который характеризует сложность вычислительной задачи. Обычно размер задачи задается либо объемом входных данных, либо значением аргумента командной строки. Интуитивно понятно, что время выполнения должно увеличиваться с размером задачи, но каждый раз при разработке и запуске программы естественно возникает вопрос, *насколько* оно увеличивается.

Еще одно качественное наблюдение: время выполнения многих программ относительно нечувствительно к объему входных данных и зависит в основном от размера задачи. Если эта взаимосвязь не выдерживается, то необходимо лучше разобраться в задаче и, возможно, лучше управлять чувствительностью времени выполнения к входным данным. Но зачастую взаимосвязь очевидна, и сейчас нашей целью будет перевод в количественную плоскость качественной зависимости времени выполнения от размера задачи.

Пример

В качестве рабочего примера мы возьмем программу `ThreeSum`, приведенную в листинге 1.4.1 — она подсчитывает количество троек в файле `N` целых чисел, которые дают в сумме 0 (без учета переполнения). Это вычисление с виду несколько надумано, но на глубинном уровне оно тесно связано со многими фундаментальными вычислительными задачами (см., например, упражнение 1.4.26).

В качестве тестовых входных данных мы возьмем файл `1Mints.txt`, который выложен на сайте книги и содержит 1 миллион случайно сгенерированных целочисленных значений. Второе, восьмое и десятое число из этого файла дают в сумме 0. Сколько еще таких троек находятся в данном файле? Программа `ThreeSum` призвана ответить на этот вопрос, но сможет ли она найти ответ за приемлемое время? Какова взаимосвязь между размером задачи `N` и временем выполнения `ThreeSum`?

В качестве первых проб попытайтесь выполнить `ThreeSum` на своем компьютере с файлами `1Kints.txt`, `2Kints.txt`, `4Kints.txt` и `8Kints.txt`, которые также находятся на сайте книги и содержат первые 1000, 2000, 4000 и 8000 чисел из `1Mints.txt` соответственно. Вы быстро получите ответ, что в файле `1Kints.txt` имеется 70 троек с нулевой суммой, а в `2Kints.txt` — 528 таких троек (рис. 1.4.1). Программе понадобится значительно больше времени на подсчет 4039 троек с нулевой суммой в файле `4Kints.txt`, а когда вы будете ожидать завершения программы для `8Kints.txt`, у вас естественно возникнет вопрос: *насколько долго будет выполняться эта программа*? Как мы вскоре увидим, ответ на этот вопрос получить несложно. В принципе часто можно довольно точно предсказать время, необходимое для выполнения программы.

Листинг 1.4.1. Насколько долго будет работать эта программа для заданного N?

```
public class  ThreeSum
{
   public static int count(int[] a)
   {  // Подсчет троек с нулевой суммой.
      int N = a.length;
      int cnt = 0;
      for (int i = 0; i < N; i++)
         for (int j = i+1; j < N; j++)
            for (int k = j+1; k < N; k++)
               if (a[i] + a[j] + a[k] == 0)
                  cnt++;
      return cnt;
   }

   public static void main(String[] args)
   {
      int[] a = In.readInts(args[0]);
      StdOut.println(count(a));
   }
}
```

```
% more 1Mints.txt
  324110
 -442472
  626686
 -157678
  508681
  123414
  -77867
  155091
  129801
  287381
  604242
  686904
 -247109
   77867
  982455
 -210707
 -922943
 -738817
   85168
  855430
  ...
```

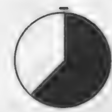
```
% java ThreeSum 1000 1Kints.txt
```




```
70
% java ThreeSum 2000 2Kints.txt
```




```
528
% java ThreeSum 4000 4Kints.txt
```




```
4039
```

Рис. 1.4.1. *Замеры времени выполнения программы*

Секундомер

Надежное измерение точного времени выполнения конкретной программы может оказаться непростым делом. К счастью, обычно хватает оценочных значений. Нам нужно просто отличать программы, которые завершаются за несколько секунд или минут, от программ, которые могут потребовать для завершения нескольких дней или месяцев, а то и больше. Ну и еще хотелось бы знать, что одна программа работает вдвое быстрее другой на той же задаче. Но все-таки нужны и точные замеры, чтобы генерировать экспериментальные данные, необходимые для формулировки и проверки правильности гипотезы о взаимосвязи времени выполнения и размера задачи. Для этого мы будем использовать тип данных `Stopwatch` (секундомер), API-интерфейс для которого приведен на рис. 1.4.2. Его метод `elapsedTime()` возвращает время, прошедшее с момента его создания (в секундах).

API-интерфейс

public class **Stopwatch**		
	`Stopwatch()`	*создание секундомера*
`double`	`elapsedTime()`	*возврат времени, прошедшего после создания*

Типичный клиент

```
public static void main(String[] args)
{
   int N = Integer.parseInt(args[0]);
   int[] a = new int[N];
   for (int i = 0; i < N; i++)
      a[i] = StdRandom.uniform(-1000000, 1000000);
   Stopwatch timer = new Stopwatch();
   int cnt = ThreeSum.count(a);
   double time = timer.elapsedTime();
   StdOut.println(cnt + " троек, " + time + " секунд");
}
```

Применение

```
% java Stopwatch 1000
51 троек, 0.488 секунд

% java Stopwatch 2000
516 троек, 3.855 секунд
```

Реализация

```
public class Stopwatch
{
   private final long start;
   public Stopwatch()
   { start = System.currentTimeMillis(); }
   public double elapsedTime()
   {
      long now = System.currentTimeMillis();
      return (now - start) / 1000.0;
   }
}
```

Рис. 1.4.2. Абстрактный тип данных для секундомера

Реализация основана на использовании системного метода `currentTimeMillis()`, который выдает текущее время в миллисекундах: сначала сохраняется время вызова конструктора, а затем, при вызове `elapsedTime()`, он вызывается еще раз, чтобы узнать прошедшее время.

Анализ экспериментальных данных

Программа `DoublingTest` из листинга 1.4.2 — более сложный клиент класса `Stopwatch`, который выдает экспериментальные данные для программы `ThreeSum`. Она генерирует последовательность случайных входных массивов, на каждом шаге удваивая размер этих массивов и выводя время выполнения `ThreeSum.count()`. Разумеется, эти эксперименты воспроизводимы: вы также можете выполнить их на своем компьютере сколько угодно раз. После запуска `DoublingTest` вы быстро обнаружите, что уже находитесь в цикле предсказания-проверки: сначала очень быстро выдаются первые несколько строк, а затем скорость появления этих строк резко замедляется.

Листинг 1.4.2. Клиент тестирования для `ThreeSum.count()`

```
public class  DoublingTest
{
  public static double timeTrial(int N)
  { // Замер времени работы ThreeSum.count() для N случайных 6-значных целых чисел.
    int MAX = 1000000;
    int[] a = new int[N];
    for (int i = 0; i < N; i++)
      a[i] = StdRandom.uniform(-MAX, MAX);
    Stopwatch timer = new Stopwatch();
    int cnt = ThreeSum.count(a);
    return timer.elapsedTime();
  }

  public static void main(String[] args)
  {  // Вывод таблицы времен выполнения.
    for (int N = 250; true; N += N)
    {  // Вывод времени для размера задачи N.
      double time = timeTrial(N);
      StdOut.printf("%7d %5.1f\n", N, time);
    }
  }
}
```

```
% java DoublingTest
   250  0.0
   500  0.0
  1000  0.1
  2000  0.8
  4000  6.4
  8000 51.1
 ...
```

Естественно, ваш компьютер отличается от нашего, и поэтому ваши значения времени выполнения будут отличаться от наших. Если, к примеру, ваш компьютер работает вдвое быстрее, вы получите значения времени выполнения, примерно вдвое меньшие наших — и можно сразу же выдвинуть очевидную гипотезу: значения времени выполнения на разных компьютерах отличаются на постоянный множитель. Но все-таки остается другой, более точный вопрос: ***сколько времени понадобится программе в виде функции от объема входных данных?*** Чтобы ответить на этот вопрос, мы представили полученные данные в виде графика. На рис. 1.4.3 приведены два графика — обычный и с логарифмическими осями; размер задачи N откладывается по оси X, а время выполнения $T(N)$ — по оси Y.

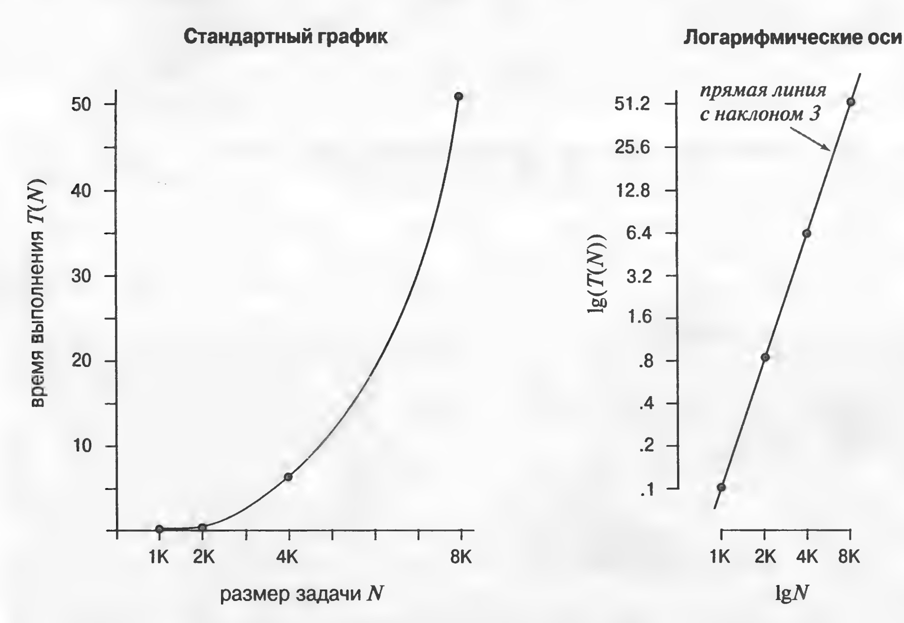


Рис. 1.4.3. Анализ экспериментальных данных (время работы `ThreeSum.count()`***)***

Логарифмический график сразу наводит на мысль о времени выполнения: точки замеров аккуратно укладываются на прямую линии с наклоном 3. Уравнение такой прямой имеет вид

$$\lg(T(N)) = 3\lg N + \lg a$$

(где a — константа), что эквивалентно

$$T(N) = aN^3$$

т.е. мы получили выражение времени выполнения в виде функции от объема входных данных. Можно взять одну точку наших данных для определения a — например, $T(8000) = 51{,}1 = a \cdot 8000^3$, откуда $a = 9{,}98 \times 10^{-11}$ — и затем использовать формулу

$$T(N) = 9{,}98 \times 10^{-11}\, N^3$$

для предсказания времени выполнения для больших N. Сейчас мы выполнили неформальную проверку гипотезы, что точки данных на графике с логарифмическими осями лежат примерно на этой прямой.

Статистические методы позволяют выполнить более тщательный анализ для определения оценки a и показателя b, но в большинстве случаев достаточно таких быстрых вычислений. Например, можно рассчитать, что на нашем компьютере для $N = 16\,000$ время выполнения будет примерно $9{,}98 \times 10^{-11}\ 16000^3 = 408{,}8$ секунд, т.е. примерно 6,8 минут (реально было замерено 409,3 секунд). Дожидаясь, пока компьютер выведет строку для $N = 16\,000$, вы можете воспользоваться этим методом для предсказания момента завершения, а затем, когда `DoublingTest` завершит работу, проверить результат.

Пока этот процесс полностью соответствует тому, как ученые пытаются понять свойства реального мира. Прямая линия на графике с логарифмическими осями эквивалентна гипотезе, что данные аппроксимируются формулой $T(N) = a\,N^b$. Такая аппроксимация называется *степенной зависимостью*. Подобные формулы описывают многие природные и искусственные явления, и вполне резонно предположить, что в их число входит и время выполнения программы. Для анализа алгоритмов существуют математические модели, которые четко поддерживают эту и аналогичные гипотезы — и мы сейчас их рассмотрим.

Математические модели

На заре развития компьютерных наук Д.Э. Кнут постулировал, что, несмотря на все сложности, которые могут возникнуть в изучении времен выполнения программ, в принципе возможно построить математическую модель, которая описывает время выполнения любой программы. Этот постулат основан на простом наблюдении: общее время выполнения программы определяется двумя основными факторами:

- стоимость выполнения каждого оператора;
- частота выполнения каждого оператора.

Первый фактор является свойством компьютера, компилятора Java и операционной системы, а второй — свойством программы и входных данных. Если в некоторой программе известны оба эти фактора, то можно просто перемножить их, просуммировать для всех инструкций программы и получить время выполнения.

Основная сложность состоит в определении частоты выполнения операторов. Некоторые операторы легко проанализировать: например, оператор, который обнуляет величину `cnt` в `ThreeSum.count()`, выполняется в точности один раз. Но для других приходится применять рассуждения более высокого уровня: например, оператор `if` в методе `ThreeSum.count()` выполняется точно

$$N\,(N-1)(N-2)\,/\,6$$

раз (количество способов, которыми можно выбрать три различных числа из входного массива — см. упражнение 1.4.1). Другие зависят от входных данных: например, количество выполнений инструкции `cnt++` в методе `ThreeSum.count()` в точности равно количеству троек с нулевой суммой во входных данных — а их может быть как ноль, так и гораздо больше. Для клиента `DoublingTest`, где числа генерируются случайным образом, можно выполнить вероятностный анализ и определить ожидаемое значение этой величины (см. упражнение 1.4.40).

Аппроксимации старшим членом

Подобные примеры анализа частоты могут приводить к сложным и длинным математическим выражениям. Например, вот приведенное выше количество выполнений оператора `if` в `ThreeSum`:

$$N(N-1)(N-2)/6 = N^3/6 - N^2/2 + N/3$$

В таких выражениях обычно слагаемые после ведущего члена относительно малы — например, для $N = 1000$ значение $-N^2/2 + N/3 \approx -499\,667$ весьма мало по сравнению с $N^3/6 \approx 166\,666\,667$ (рис. 1.4.4). Чтобы игнорировать относительно мелкие слагаемые и, таким образом, существенно упрощать математические формулы трудоемкости, мы будем часто использовать знак *тильды* (~). Он позволяет работать с *аппроксимациями старшим членом* (табл. 1.4.1), когда отбрасываются слагаемые более низкого порядка, которые вносят незначительный вклад в интересующие нас величины и только усложняют формулы:

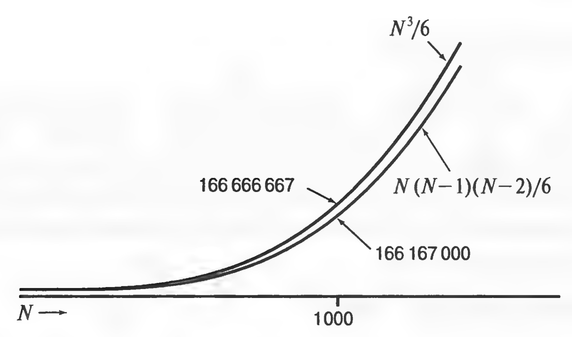


Рис. 1.4.4. *Аппроксимация старшим членом*

Таблица 1.4.1. Типичные аппроксимации

Функция	Аппроксимация старшим членом	Порядок роста
$N^3/6 - N^2/2 + N/3$	$\sim N^3/6$	N^3
$N^2/2 - N/2$	$\sim N^2/2$	N^2
$\lg N + 1$	$\sim \lg N$	$\lg N$
3	~ 3	1

Определение. Мы будем использовать запись $\sim f(N)$ для обозначения любой функции, которая, будучи разделенной на $f(N)$, приближается к 1 при увеличении N. Мы будем использовать запись $g(N) \sim f(N)$ для обозначения того, что $g(N)/f(N)$ приближается к 1 при увеличении N.

Например, мы используем аппроксимацию $\sim N^3/6$ для описания количества выполнений оператора `if` в программе `ThreeSum`: ведь $N^3/6 - N^2/2 + N/3$, разделенное на $N^3/6$, при увеличении N стремится к 1. Чаще всего встречаются аппроксимации вида $g(N) \sim af(N)$, где $f(N) = N^b(\log N)^c$, a, b и c — константы, а $f(N)$ называется *порядком роста* $g(N)$. При использовании логарифма основание обычно не указывается, т.к. кон-

станта a поглощает разницу. Эта обобщенная формула охватывает относительно небольшое количество функций, которые обычно встречаются при изучении порядка роста времен выполнения программ (см. табл. 1.4.2), за исключением экспоненциальной функции, о которой мы поговорим в главе 6. Мы рассмотрим эти функции несколько более подробно, а также кратко обсудим, почему они возникают в анализе алгоритмов, после того как завершим исследование программы `ThreeSum`.

Таблица 1.4.2. Часто встречающиеся функции порядка роста

Описание	Функция
Константа	1
Логарифмическая	$\log N$
Линейная	N
Линейно-логарифмическая	$N\log N$
Квадратичная	N^2
Кубическая	N^3
Экспоненциальная	2^N

Примерное время выполнения

Чтобы соответствовать подходу, предложенному Кнутом, для вывода математического выражения для общего времени выполнения Java-программы можно (в принципе) узнать, в какие машинные инструкции компилятор Java переводит каждый оператор Java, и взять из машинных спецификаций время выполнения каждой машинной инструкции — так можно получить общее итоговое время. Этот процесс для программы `ThreeSum` вкратце изображен на рис. 1.4.5.

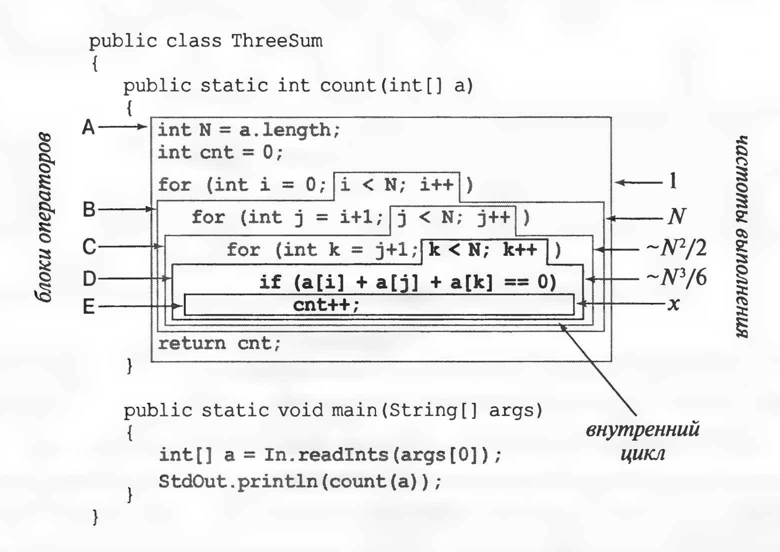


Рис. 1.4.5. Анализ частот выполнения операторов программы

Мы разграничиваем блоки операторов Java по частоте их выполнения, выводим аппроксимации старшими членами для этих частот, определяем стоимость каждого оператора, после чего вычисляем общую трудоемкость. Правда, некоторые частоты могут зависеть от входных данных. В нашем случае количество выполнений оператора `cnt++`, несомненно, зависит от входных данных: это количество троек с нулевыми суммами, а оно может быть любым от 0 до $\sim N^3/6$. Мы не будем вдаваться в детали (значения констант) для каждой конкретной системы, только подчеркнем, что константы t_0, t_1, t_2, ... для времен выполнения блоков операторов означают, что каждый блок Java-операторов соответствует машинным инструкциями, которые требуют для своего выполнения фиксированного времени. Самое важное в этом примере — наблюдение, что в окончательном результате важны лишь инструкции, которые выполняются чаще всего. Эти инструкции мы будем называть ***внутренним циклом*** программы. В программе `ThreeSum` внутренний цикл — это операторы, которые увеличивают значение `k` и проверяют, меньше ли оно `N`, а также операторы проверки суммы трех чисел на равенство нулю (и, возможно, оператор увеличения счетчика, но это зависит от входных данных). Такое поведение типично: значения времени выполнения очень многих программ зависят от небольшого подмножества их инструкций.

Гипотеза порядка роста

И эксперименты, приведенные в листинге 1.4.2 и показанные на рис. 1.4.3, и математическая модель, приведенная в табл. 1.4.3, поддерживают описанную ниже гипотезу.

Таблица 1.4.3. Анализ времени выполнения программы (пример)

Блок операторов	Время в секундах	Частота	Общее время
E	t_0	x (зависит от входных данных)	$t_0 x$
D	t_1	$N^3/6 - N^2/2 + N/3$	$t_1\ (N^3/6 - N^2/2 + N/3)$
C	t_2	$N^2/2 - N/2$	$t_2\ (N^2/2 - N/2)$
B	t_3	N	$t_3\ N$
A	t_4	1	t_4
		итого	$(t_1/6)\ N^3 + (t_2/2 - t_1/2)\ N^2 + (t_1/3 - t_2/2 + t_3)\ N + t_4 + t_0 x$
		аппроксимация старшим членом	$\sim (t_1/6)\ N^3$ (для малых x)
		порядок роста	N^3

Свойство А. Порядок роста для времени выполнения программы `ThreeSum` (определение количества троек с нулевой суммой из N чисел) равен N^3.

Обоснование. Пусть $T(N)$ — время выполнения программы `ThreeSum` для N чисел. Описанная выше математическая модель показывает, что $T(N) \sim aN^3$ для некоторой зависящей от процессора константы a; эксперименты на многих компьютерах подтверждают эту аппроксимацию.

Во всей книге мы будем использовать слово *свойство* для обозначения гипотезы, которую необходимо экспериментально проверить. Результат нашего математического анализа в точности совпадает с результатом экспериментов: время работы программы ThreeSum равно aN^3 для некоторой зависящей от процессора константы a. Это соответствие удостоверяет и эксперименты, и математическую модель, а также позволяет глубже понять свойства программы, т.к. для определения показателя степени эксперименты не обязательны. Приложив некоторые усилия, мы могли бы определить и значение a для какой-то конкретной системы, но этим видом деятельности обычно занимаются эксперты в ситуациях, когда производительность критична.

Анализ алгоритмов

Гипотезы наподобие свойства А важны тем, что они соединяют абстрактный мир Java-программ с реальным миром компьютеров, на которых они выполняются. Работа с порядком величины позволяет выполнить еще один шаг — отделить программу от алгоритма, который она реализует. Свойство того, что время выполнения ThreeSum имеет порядок роста N^3, не зависит от того, что эта программа реализована на Java, или от оборудования, на котором она выполняется — это может быть ваш ноутбук, чей-то мобильный телефон или оборонный суперкомпьютер. Это время зависит в основном от того, что программа просматривает все различные тройки входных чисел. Порядок роста определяется используемым *алгоритмом* (и иногда моделью ввода). Отделение алгоритма от реализации на конкретном компьютере — мощная концепция, которая позволяет получить знание о производительности алгоритмов и применять это значение на любом компьютере. Например, можно сказать, что программа ThreeSum представляет собой реализацию примитивного алгоритма: вычисляются суммы всех возможных троек и подсчитываются те тройки, которые дают в сумме ноль. Мы ожидаем, что реализация этого алгоритма на любом языке программирования на любом компьютере даст время выполнения, пропорциональное N^3. Вообще говоря, значительный объем знаний о производительности классических алгоритмов был разработан еще несколько десятков лет назад, но эти знания относятся и к современным компьютерам.

Модель стоимости

При рассмотрении свойств алгоритмов основное внимание мы уделяем *модели стоимости*, которая определяет базовые действия, выполняемые алгоритмом для решения задачи. Например, приведенная ниже на врезке модель стоимости для задачи поиска нулевых троек — это количество обращений к элементам массива. С помощью такой модели можно сформулировать точные математические утверждения о свойствах алгоритма, а не конкретной реализации.

Утверждение Б. Примитивный алгоритм для решения задачи поиска нулевых троек выполняет $\sim N^3/2$ обращений к массиву из N чисел.

Доказательство. Алгоритм обращается к каждому из трех чисел для каждой из $\sim N^3/6$ троек.

Модель стоимости для задачи поиска нулевых троек. При изучении алгоритмов для решения задачи поиска нулевых троек мы подсчитываем *обращения к массиву* (количество случаев доступа к элементам массива, как для чтения, так и для записи).

Слово *утверждение* мы будем использовать для формулировки математических высказываний об алгоритмах в терминах модели стоимости. Во всей книге мы будем рас-

сматривать алгоритмы в рамках какой-то конкретной модели стоимости. Мы хотим наглядно показать, что порядок роста времени выполнения для конкретной реализации совпадает с порядком роста стоимости лежащего в основе этой реализации алгоритма (т.е. модель стоимости должна содержать операции, которые находятся во внутреннем цикле). Мы будем искать точные математические формулы для алгоритмов (утверждения), а также гипотезы о производительности реализаций (свойства), которые можно экспериментально проверить. В рассматриваемом случае утверждение Б математически обосновывает гипотезу, которая была предложена в свойстве А и подтверждена экспериментами — в соответствии с научным методом.

Сводка

Для многих программ разработка математической модели времени выполнения осуществляется с помощью следующих шагов.

- Разработка *модели ввода* — включая и определение размера задачи.
- Выявление *внутреннего цикла*.
- Определение *модели стоимости*, которая включает в себя операции во внутреннем цикле.
- Определение частоты выполнения этих операций для конкретных входных данных. Для этого может потребоваться математический *анализ*, и мы ниже в данной книге рассмотрим несколько примеров в контексте конкретных фундаментальных алгоритмов.

Если программа оформлена в виде совокупности методов, мы обычно рассматриваем такие методы отдельно. В качестве примера можно предложить нашу демонстрационную программу `BinarySearch` из раздела 1.1.

Бинарный поиск

Модель входных данных — массив `a[]` размером N; *внутренний цикл* — операторы единственного цикла `while`; *модель стоимости* — операция сравнения (сравнение значений двух элементов массива); а *анализ*, начатый в разделе 1.1 и более подробный в утверждении Б в разделе 3.1, показывает, что количество сравнений не превышает $\lg N + 1$.

Белый список

Модель входных данных — N чисел из белого списка и M чисел из стандартного ввода ($M >> N$); *внутренний цикл* — операторы единственного цикла `while`; *модель стоимости* — операция сравнения (унаследованная от бинарного поиска); а *анализ* непосредственно следует из анализа бинарного поиска: количество сравнений не превышает $M(\lg N + 1)$.

Отсюда можно сделать вывод, что порядок роста времени обработки белого списка не превышает $M\lg N$.

- При малых N может доминировать стоимость ввода-вывода.
- Количество сравнений зависит от входных данных и находится в пределах от $\sim M$ до $\sim M\lg N$ — в зависимости от количества чисел в белом списке и длительности поиска в нем (обычно $\sim M\lg N$).
- Мы предполагаем, что стоимость выполнения `Arrays.sort()` невелика по сравнению с $M\lg N$. Метод `Arrays.sort()` реализует *сортировку слиянием*, а в разделе 2.2 будет показано, что порядок роста времени выполнения сортировки слиянием равен $N\lg N$ (см. утверждение G в главе 2), поэтому такое предположение верно.

Таким образом, модель поддерживает нашу гипотезу из раздела 1.1: *алгоритм бинарного поиска* позволяет выполнять вычисления и при больших M и N. При удвоении длины потока стандартного ввода можно ожидать удвоения времени выполнения; при удвоении размера белого списка можно ожидать лишь незначительного увеличения времени выполнения.

Разработка математических моделей для анализа алгоритмов — плодотворная область исследований, которая не совсем вписывается в рамки данной книги. Но, как вы убедитесь при изучении бинарного поиска, сортировки слиянием и многих других алгоритмов, понимание некоторых математических моделей абсолютно необходимо для понимания эффективности фундаментальных алгоритмов, поэтому мы зачастую будем вдаваться в математические детали и/или приводить готовые результаты классических исследований. При этом нам будут встречаться различные функции и аппроксимации, которые широко используются в математическом анализе. Для справки мы приведем некоторые из них в табл. 1.4.4 и 1.4.5.

Таблица 1.4.4. Функции, которые часто встречаются при анализе алгоритмов

Описание	Обозначение	Определение
Округление в меньшую сторону	$\lfloor x \rfloor$	Наибольшее целое число, которое не больше x
Округление в большую сторону	$\lceil x \rceil$	Наименьшее целое число, которое не меньше x
Натуральный логарифм	$\ln N$	$\log_e N$ (такое x, что $e^x = N$)
Двоичный логарифм	$\lg N$	$\log_2 N$ (такое x, что $2^x = N$)
Целый двоичный логарифм	$\lfloor \lg N \rfloor$	Наибольшее целое число, которое не больше $\lg N$ (количество битов в двоичном представлении N) -1
Гармонические числа	H_N	$1 + 1/2 + 1/3 + 1/4 + \ldots + 1/N$
Факториал	$N!$	$1 \times 2 \times 3 \times 4 \times \ldots \times N$

Таблица 1.4.5. Аппроксимации, полезные при анализе алгоритмов

Описание	Аппроксимация
Гармоническая сумма	$H_N = 1 + 1/2 + 1/3 + 1/4 + \ldots + 1/N \sim \ln N$
Треугольная сумма	$1 + 2 + 3 + 4 + \ldots + N \sim N^2/2$
Геометрическая сумма	$1 + 2 + 4 + 8 + \ldots + N = 2N - 1 \sim 2N$ при $N = 2^n$
Аппроксимация Стирлинга	$\lg N! = \lg 1 + \lg 2 + \lg 3 + \lg 4 + \ldots + \lg N \sim N \lg N$
Биномиальные коэффициенты	$\binom{N}{k} \sim N^k/k!$, если k — небольшая константа
Экспонента	$(1 - 1/x)^x \sim 1/e$

Классификация порядков роста

Для реализации алгоритмов используется небольшое количество структурных примитивов — операторы, условия, циклы, вложенность и вызовы методов — поэтому обычно порядок роста стоимости выражается одной из немногих функций от размера задачи N. Эти функции — а также их названия, типичный код, который приводит к их появлению, и примеры — приведены в табл. 1.4.6.

Таблица 1.4.6. Сводка распространенных гипотез для порядка роста

Описание	Порядок роста	Типичный код	Описание	Пример
Константный	1	`a = b + c;`	Оператор	Сложение двух чисел
Логарифмический	$\log N$	см. листинг 1.1.4	Деление пополам	Двоичный поиск
Линейный	N	`double max = a[0];` `for (int i = 1; i < N; i++)` `  if (a[i] > max) max = a[i];`	Цикл	Поиск максимума
Линейно-логарифмический	$N\log N$	см. алгоритм 2.4	Разделяй и властвуй	Сортировка слиянием
Квадратичный	N^2	`for (int i = 0; i < N; i++)` `  for (int j = i+1; j < N; j++)` `    if (a[i] + a[j] == 0)` `      cnt++;`	Двойной цикл	Проверка всех пар
Кубический	N^3	`for (int i = 0; i < N; i++)` `  for (int j = i+1; j < N; j++)` `    for (int k = j+1; k < N; k++)` `      if (a[i] + a[j] + a[k] == 0)` `        cnt++;`	Тройной цикл	Проверка всех троек
Экспоненциальный	2^N	см. главу 6	Исчерпывающий поиск	Проверка всех подмножеств

Константный

Программа, порядок роста времени выполнения которой равен *константе* (постоянен), выполняет фиксированное количество операций для завершения своей работы — поэтому время ее выполнения не зависит от N. Большинство операций в Java требуют постоянного времени.

Логарифмический

Программа с *логарифмическим* порядком роста времени выполнения работает несколько медленнее программы с постоянным временем выполнения. Классический пример программы с логарифмическим временем выполнения относительно размера задачи — *бинарный поиск* (см. программу `BinarySearch` в листинге 1.1.4). Основание логарифма не важно по сравнению с порядком роста (т.к. все логарифмы с постоянным основанием отличаются на постоянный коэффициент), поэтому при указании порядка роста мы пишем просто $\log N$.

Линейный

Программы с постоянной трудоемкостью обработки любой части входных данных или основанные на единственном цикле `for`, встречаются довольно часто. Порядок роста у таких программ называется *линейным*: их время выполнения пропорционально N.

Линейно-логарифмический

Термин *линейно-логарифмический* служит для описания программ, время выполнения которых для задач размером N имеет порядок роста $N\log N$. Здесь основание логарифма также не имеет отношения к порядку роста. Характерный пример линейно-логарифмического алгоритма — методы `Merge.sort()` (см. алгоритм 2.4) и `Quick.sort()` (см. алгоритм 2.5).

Квадратичный

Типичная программа со временем выполнения порядка N^2 содержит два вложенных цикла `for` для каких-то вычислений, выполняемых для всех пар из N элементов. Прототипами подобных программ могут служить элементарные алгоритмы сортировки `Selection.sort()` (см. алгоритм 2.1) и `Insertion.sort()` (см. алгоритм 2.2).

Кубический

Типичная программа со временем выполнения порядка N^3 содержит три вложенных цикла `for` для каких-то вычислений, выполняемых для всех троек из N элементов. Прототипом может служить пример из данного раздела — программа `ThreeSum`.

Экспоненциальный

В главе 6 (но не раньше!) мы рассмотрим программы, время выполнения которых пропорционально 2^N, а то и больше. Мы будем использовать термин *экспоненциальный* для алгоритмов, порядок роста которых равен b^N для любой константы $b > 1$, хотя разные значения b дают весьма различающиеся значения времени. Экспоненциальные алгоритмы работают невероятно медленно и попросту не заканчиваются для больших задач. Однако они играют важную роль в теории алгоритмов, т.к. существует большой класс задач, для которых не известно ничего лучше экспоненциальных методов решения.

Эта классификация наиболее распространена, но она, несомненно, не полна. Порядок роста стоимости алгоритма может быть равен $N^2\log N$, или $N^{3/2}$, или какой-то аналогичной функции. В принципе подробный анализ алгоритмов может потребовать весь спектр математических средств, разработанных на протяжении последних столетий.

Подавляющее большинство рассматриваемых нами алгоритмов обладают очевидными характеристиками производительности, которые можно описать одним из приведенных выше порядков роста. Соответственно, обычно мы можем работать с конкретными утверждениями относительно модели стоимости — например, что *сортировка слиянием выполняет от* $^1/_2 N\log N$ *до* $N\log N$ *сравнений*, откуда сразу следуют гипотезы (свойства) вроде *сортировка слиянием имеет линейно-логарифмический порядок роста времени выполнения*. Чтобы не быть многословными, мы обычно будем выражать такие заявления в следующем виде: *сортировка слиянием имеет линейно-логарифмическую сложность*.

Графики на рис. 1.4.6 демонстрируют практическую важность порядка роста. По оси X откладывается размер задачи, а по оси Y — время выполнения. Из графиков видно, что квадратичные и кубические алгоритмы не пригодны для решения больших задач. Оказывается, для некоторых важных задач существуют с виду естественные решения с квадратичной сложностью, но имеются и хитроумные линейно-логарифмические алгоритмы. Такие алгоритмы (в том числе и сортировка слиянием) очень важны в практическом плане, т.к. они позволяют работать с задачами, гораздо большими, чем разрешимые квадратичными методами. И, естественно, в данной книге мы будем рассматривать в первую очередь логарифмические, линейные и линейно-логарифмические алгоритмы решения фундаментальных задач.

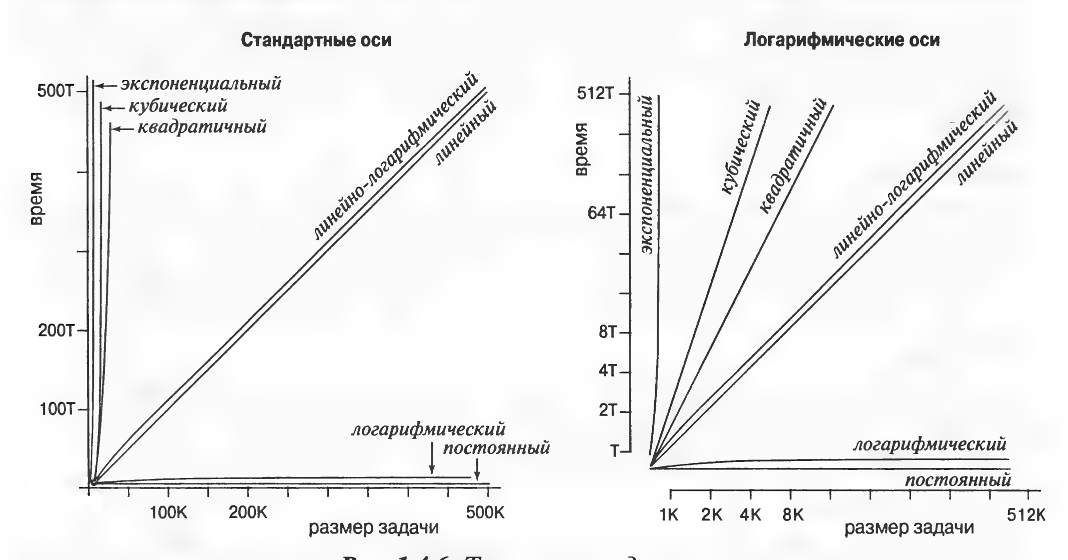


Рис. 1.4.6. Типичные порядки роста

Проектирование быстрых алгоритмов

Одна из основных причин изучения порядка роста для времени выполнения программ — помощь в проектировании более быстрых алгоритмов для решения той же задачи. Для иллюстрации мы сейчас рассмотрим более быстрый алгоритм решения задачи поиска троек с нулевой суммой. Как можно сочинить более быстрый алгоритм, даже не

приступив к изучению алгоритмов? Ответ таков: мы *уже* рассмотрели и применяли два классических алгоритма — *сортировку слиянием* и *бинарный поиск* — и знаем, что сортировка слиянием имеет линейно-логарифмическую сложность, а бинарный поиск — логарифмическую. Как можно воспользоваться этими знаниями для решения задачи поиска троек?

Разминка: поиск пар чисел

Рассмотрим более простую задачу — определение количества *пар* целых чисел во входном файле, сумма которых равна нулю. Для еще большего упрощения предположим, что все числа различны. У этой задачи имеется простое решение с квадратичным временем выполнения: достаточно убрать из метода `ThreeSum.count()` цикл по `k` и обращения к `a[k]` и оставить только двойной цикл проверки всех пар (см. табл. 1.4.6) — назовем эту реализацию `TwoSum`. Реализация в листинге 1.4.3 показывает, как сортировка слиянием и бинарный поиск (см. листинг 1.1.4) могут служить основой для *линейно-логарифмического* решения задачи поиска пар чисел. Усовершенствованный алгоритм основан на том факте, что элемент `a[i]` принадлежит какой-то нулевой паре тогда и только тогда, когда значение `-a[i]` присутствует в массиве (и `a[i]` не равно нулю). Для решения задачи мы отсортируем массив (чтобы мог работать бинарный поиск), а затем для каждого элемента массива `a[i]` выполним бинарный поиск значения `-a[i]` с помощью метода `rank()` из `BinarySearch`. Если получен индекс `j` такой, что `j > i`, к счетчику добавляется единица.

Листинг 1.4.3. Линейно-логарифмическое решение для задачи поиска нулевых пар

```
import java.util.Arrays;
public class TwoSumFast
{
   public static int count(int[] a)
   {  // Подсчет пар чисел с нулевой суммой.
      Arrays.sort(a);
      int N = a.length;
      int cnt = 0;
      for (int i = 0; i < N; i++)
         if (BinarySearch.rank(-a[i], a) > i)
            cnt++;
      return cnt;
   }
   public static void main(String[] args)
   {
      int[] a = In.readInts(args[0]);
      StdOut.println(count(a));
   }
}
```

В проверке должны рассматриваться три описанных ниже случая.

- Неудачный бинарный поиск возвращает `-1`, и счетчик увеличивать не надо.
- Бинарный поиск возвращает `j > i`, тогда `a[i] + a[j] = 0`, и нужно увеличить счетчик.
- Бинарный поиск возвращает `j` между `0` и `i`, тогда также `a[i] + a[j] = 0`, но счетчик увеличивать не надо, т.к. этот случай уже был подсчитан раньше.

Результат вычисления в точности тот же, что и раньше, но он требует гораздо меньшего времени. Время выполнения сортировки слиянием пропорционально $N\log N$, и каждый из N бинарных поисков требует времени, пропорционального $\log N$ — поэтому время выполнения всего алгоритма пропорционально $N\log N$. Разработка такого скоростного алгоритма является не просто теоретическим упражнением: он позволяет решать задачи гораздо большего размера. Например, на вашем компьютере нетрудно решить задачу поиска пар для миллиона целых чисел (`1Mints.txt`) за приемлемое время, однако вам пришлось бы очень долго ждать завершения квадратичного алгоритма (см. упражнение 1.4.41).

Быстрый алгоритм для троек

Этот же принцип применим и для задачи подсчета троек. Предположим также, что все числа различны. Пара чисел `a[i]` и `a[j]` является частью тройки с нулевой суммой тогда и только тогда, когда значение `-(a[i] + a[j])` присутствует в массиве (и не равно `a[i]` или `a[j]`). Код в листинге 1.4.4 сортирует массив, а затем выполняет $N(N-1)/2$ бинарных поисков, каждый из которых выполняется за время, пропорциональное $\log N$ — общее время выполнения получается пропорциональным $N^2\log N$. Обратите внимание, что в этом случае стоимость сортировки уже не важна. Это решение также позволяет решать задачи гораздо большего размера (см. упражнение 1.4.42). Графики на рис. 1.4.7 демонстрируют различие стоимостей этих четырех алгоритмов для уже рассмотренных размеров задач. Конечно, такие различия представляют собой серьезную причину для поиска более быстрых алгоритмов.

Листинг 1.4.4. Решение сложности $N^2\log N$ для задачи поиска нулевых троек

```
import java.util.Arrays;
public class ThreeSumFast
{
   public static int count(int[] a)
   {  // Подсчет троек с нулевой сумой.
      Arrays.sort(a);
      int N = a.length;
      int cnt = 0;
      for (int i = 0; i < N; i++)
         for (int j = i+1; j < N; j++)
            if (BinarySearch.rank(-a[i]-a[j], a) > j)
               cnt++;
      return cnt;
   }

   public static void main(String[] args)
   {
      int[] a = In.readInts(args[0]);
      StdOut.println(count(a));
   }
}
```

Нижние границы

В табл. 1.4.7 приведена сводка информации из данного раздела. Сразу же возникает интересный вопрос: можно ли для задач поиска пар и троек найти алгоритмы, которые работают значительно быстрее, чем `TwoSumFast` и `ThreeSumFast`?

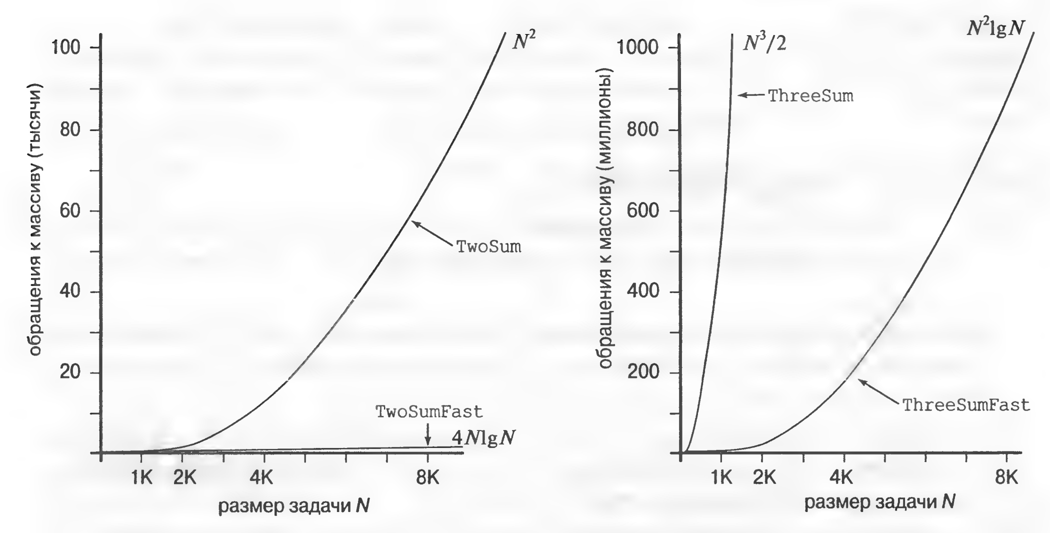


Рис. 1.4.7. Стоимости алгоритмов для решения задач поиска пар и троек

Существует ли линейный алгоритм для пар или линейно-логарифмический для троек? Ответы на эти вопросы таковы: *нет* — для пар (в модели, в которой выполняется подсчет и сравнения линейных или квадратичных функций от чисел) и *неизвестно* — для троек, хотя эксперты считают, что наилучший из возможных алгоритм для троек обладает квадратичной сложностью. Идея определения нижней границы для порядка роста времени выполнения в худшем случае для всех возможных алгоритмов решения задачи очень привлекательна, и мы еще вернемся к ней в разделе 2.2 при рассмотрении сортировки. Нетривиальные нижние границы бывает очень трудно сформулировать, но они очень полезны при поиске эффективных алгоритмов.

Таблица 1.4.7. Сводка значений времени выполнения для программ поиска пар и троек

Алгоритм	Порядок роста времени выполнения
`TwoSum`	N^2
`TwoSumFast`	$N\log N$
`ThreeSum`	N^3
`ThreeSumFast`	$N^2\log N$

Примеры в этом разделе подготавливают почву для изучения алгоритмов в данной книге. Во всей книге наша стратегия при рассмотрении новой задачи будет выглядеть так, как описано ниже.

- Реализуем и анализируем простое и очевидное решение задачи. Такие решения, вроде `ThreeSum` и `TwoSum`, мы будем называть *примитивными* решениями или решениями "в лоб" (brute-force).
- Рассматриваем алгоритмические усовершенствования, обычно предназначенные для снижения порядка роста времени выполнения — такие как `TwoSumFast` и `ThreeSumFast`.
- Экспериментально проверяем предположение, что новые алгоритмы работают быстрее.

Во многих случаях мы будем рассматривать *несколько* алгоритмов для одной и той же задачи, т.к. время выполнения — лишь один из критериев при выборе алгоритма для практической задачи. Мы будем рассматривать этот принцип в контексте фундаментальных задач на протяжении всей книги.

Эксперименты с удвоением

Ниже представлен простой и эффективный способ для прогноза производительности и для определения приблизительного порядка роста времени выполнения любой программы.

- Напишите генератор входных данных, который выдает данные, похожие на возникающие на практике (такие как случайные числа в методе `timeTrial()` в `DoublingTest`).
- Выполните программу `DoublingRatio`, приведенную в листинге 1.4.5 — это модификация `DoublingTest`, которая вычисляет отношение времени выполнения данного прогона к предыдущему.
- Повторяйте так, пока не достигнете сходимости к предельному значению 2^b.

Листинг 1.4.5. Программа DoublingRatio (для выполнения экспериментов)

```
public class DoublingRatio
{
   public static double timeTrial(int N)
   // то же, что и в DoublingTest (листинг 1.4.2)

   public static void main(String[] args)
   {
      double prev = timeTrial(125);
      for (int N = 250; true; N += N)
      {
         double time = timeTrial(N);
         StdOut.printf("%6d %7.1f ", N, time);
         StdOut.printf("%5.1f\n", time/prev);
         prev = time;
      }
   }
}
```

Результаты экспериментов

```
% java DoublingRatio
  250     0.0   2.7
  500     0.0   4.8
 1000     0.1   6.9
 2000     0.8   7.7
 4000     6.4   8.0
 8000    51.1   8.0
```

Прогнозы

```
16000    408.8   8.0
32000   3270.4   8.0
64000  26163.2   8.0
```

Этот тест неэффективен, если отношения не сходятся к предельному значению, но для очень многих программ они все-таки сходятся, из чего можно сделать следующие выводы.

- Порядок роста времени выполнения примерно равен N^b.
- Для прогноза времени выполнения умножьте последнее замеренное время выполнения на 2^b и удвойте N, и повторяйте так до получения нужного значения N. Если прогноз необходим для размера входных данных, который не равен 2 в степени N, можно рассчитать соответствующие значения (см. упражнение 1.4.9).

Как видно из результатов работы программы `DoublingRatio`, отношение для программы `ThreeSum` приблизительно равно 8, и это позволяет предсказать значения времени выполнения для N = 16 000, 32 000, 64 000 — они равны `408.8`, `3270.4` и `26163.2` секунд соответственно. Для этого нужно просто время выполнения для N = 8 000 (51,1) несколько раз умножить на 8.

Этот тест примерно эквивалентен процессу, описанному в подразделе "Анализ экспериментальных данных" выше в настоящем разделе — выполнение экспериментов, нанесение точек на график с логарифмическими осями, формулировка гипотезы, что время выполнения равно aN^b, определение значения b, исходя из наклона линии, а затем аналитическое определение a — но его проще применять. Чтобы вручную точно получить прогноз времени выполнения, достаточно выполнить программу `DoublingRatio`. Когда отношение сойдется к предельному значению, просто умножайте на это отношение, чтобы заполнять следующие значения в таблице. Ваша приблизительная модель порядка роста — это степенная зависимость с двоичным логарифмом этого отношения в показателе степени.

Почему это отношение сходится к константе? Простое математическое вычисление показывает, что это должно быть верно для всех распространенных и уже рассмотренных порядков роста (кроме экспоненциального):

Утверждение В (удвоение). Если $T(N) \sim aN^b \lg N$, то $T(2N) \,/\, \mathrm{T}(N) \sim 2^b$.

Доказательство. Это непосредственно вытекает из следующего вычисления:

$$\begin{aligned} T(2N) \,/\, \mathrm{T}(N) &= a(2N)^b \lg(2N) \,/\, aN^b \lg N \\ &= 2^b\,(1 + \lg 2 \,/\, \lg N) \\ &\sim 2^b \end{aligned}$$

В общем случае логарифмический множитель нельзя игнорировать при разработке математической модели, но в предсказании производительности для гипотезы удвоения он играет менее важную роль.

Эксперименты с отношением при удвоении следует проводить для всех новых программ, для которых важны вопросы производительности — это очень простой способ для оценки порядка роста времени выполнения; при этом может оказаться, что из-за какой-то ошибки алгоритма программа работает не так эффективно, как предполагалось. В общем случае можно использовать гипотезы о порядке роста времени выполнения для предсказания производительности программы одним из описанных ниже способов.

Оценка разрешимости в случае больших задач

Для каждой новой программы важен ответ на следующий основной вопрос: *сможет ли программа обработать необходимый объем входных данных за приемлемое время?*

Для ответа на подобные вопросы для больших объемов данных мы выполняем экстраполяцию с гораздо большим отношением (например, 10), как показано в четвертом столбце табл. 1.4.8. Многие пользователи — банкиры, ежедневно анализирующие финансовые модели, ученые, обрабатывающие экспериментальные данные, или инженеры, моделирующие проверку своего проекта — довольно часто запускают программы, которые выполняются по нескольку часов, и в таблице как раз отражена такая ситуация. Зная порядок роста для времени выполнения алгоритма, можно получить точную информацию, необходимую для понимания ограничений на размер задачи, который доступен для выполнения. *Выработка такого понимания — наиболее важная причина изучения производительности алгоритмов.* Без него у вас не будет ни малейшего представления, сколько времени может понадобиться программе, а с ним вы сможете быстро прикинуть на любом клочке бумаги стоимость вычислений и принять соответствующее решение.

Таблица 1.4.8. Прогнозы на основе порядка роста

Порядок роста времени				для программы, которая работает несколько часов для входных данных размера N	
Описание	**Функция**	**Отношение 2**	**Отношение 10**	**Прогноз времени для $10N$**	**Прогноз времени для $10N$ на компьютере в 10 раз быстрее**
Линейный	N	2	10	Один день	Несколько часов
Линейно-логарифмический	$N\log N$	2	10	Один день	Несколько часов
Квадратичный	N^2	4	100	Несколько недель	Один день
Кубический	N^3	8	1000	Несколько месяцев	Несколько недель
Экспоненциальный	2^N	2^N	2^{9N}	Никогда	Никогда

Оценка перехода на более быстрый компьютер

Иногда может возникать и другой, не менее важный, вопрос: *насколько быстрее будет решена задача, если решать ее на более быстром компьютере?* В общем случае, если новый компьютер работает в x раз быстрее старого, то время выполнения уменьшится в x раз. Однако обычно важен совсем другой вопрос: можно ли решать на новом компьютере задачи большего размера? Как повлияет переход на другой компьютер на время выполнения? Ответ на этот вопрос также можно получить на основе порядка роста.

Знаменитое эмпирическое правило, известное как *закон Мура*, гласит, что каждые 18 месяцев скорость и объем памяти компьютеров удваиваются, а каждые пять лет становятся примерно в 10 раз больше. В табл. 1.4.8 видно, что закон Мура отстает от квадратичных и кубических алгоритмов, а тесты показывают, что через 18 месяцев объем входных данных можно увеличить в два раза, но не в 4 или 8.

Предостережения

Существует множество факторов, которые могут помешать получить согласованные и верные результаты при тщательном анализе производительности программ. Все они приводят к мысли, что одно или несколько базовых предположений, на которых основа-

на та или иная гипотеза, не вполне верны. Можно выдвинуть новые гипотезы на основе новых предположений, но чем больше деталей приходится учитывать, тем аккуратнее следует относиться к анализу.

Большие константы

В аппроксимациях старшим членом мы игнорируем постоянные коэффициенты при слагаемых с меньшими степенями, но это не всегда правильно. Например, аппроксимируя функцию $2N^2 + cN$ как $\sim 2N^2$, мы предполагаем, что константа c невелика. Если это не так (например, c равно 10^3 или 10^6), аппроксимация окажется ошибочной. Поэтому следует учитывать возможность больших констант.

Не доминирующий внутренний цикл

Предположение, что внутренний цикл вносит в сложность основной вклад, не всегда верно. В модели сложности может рассматриваться не настоящий внутренний цикл, или размер задачи может быть недостаточно большим, чтобы старший член в математическом описании частоты выполнения инструкций существенно превысил младшие члены, давая возможность их игнорировать. Некоторые программы выполняют существенный объем кода за рамками внутреннего цикла, и его также следует учитывать. То есть модель стоимости может потребовать пересмотра.

Время выполнения инструкций

Предположение, что каждая инструкция всегда требует одного и того же времени, не всегда верно. Например, в большинстве современных вычислительных систем используется *кеширование* для организации памяти, из-за чего доступ к элементам в крупных массивах может потребовать гораздо больше времени, если они не находятся рядом. Эффект кеширования можно наблюдать на примере программы `ThreeSum`: просто позвольте программе `DoublingTest` поработать несколько дольше. После достижения сходимости к 8 отношение времен выполнения для больших массивов может резко увеличиться — как раз из-за кеширования.

Особенности системы

Обычно в компьютере выполняется много разных процессов. Java — лишь одно из приложений, претендующих на ресурсы, и в самой системе Java имеется множество параметров и управляющих элементов, которые существенно влияют на производительность. Сборщик мусора, JIT-компилятор или загрузка из Интернета могут значительно повлиять на результаты экспериментов. Эти факторы нарушают базовый принцип научного метода, что эксперименты должны быть воспроизводимыми, т.к. то, что происходит в вашем компьютере сейчас, невозможно воспроизвести снова. Все, что творится в системе, *в принципе* должно иметь пренебрежимо малый эффект, либо должен существовать способ управления этими процессами.

Почти ничья

Часто при сравнении двух различных программ для решения одной и той же задачи первая может работать быстрее в одних ситуациях, а вторая — в других. На разницу могут повлиять один или несколько вышеописанных факторов. Некоторые программисты (и студенты) просто лезут из кожи вон, чтобы найти "наилучшую" реализацию, но подобную работу лучше оставить экспертам.

Сильная зависимость от входных данных

Одно из первых предположений, которое лежит в основе определения порядка роста для времени выполнения программы — это что время работы относительно нечувствительно к входным данным. Если это не так, мы можем получить невразумительные результаты или не сумеем проверить гипотезу. Например, допустим, что программа `ThreeSum` модифицирована так, чтобы ответить на вопрос: *содержат ли входные данные хотя бы одну тройку с нулевой суммой*? Для этого нужно сделать возвращаемое значение логическим, заменить `cnt++` на `return true` и добавить в конец оператор `return false`. Порядок роста времени выполнения для полученной программы будет *постоянным*, если первые же три числа дают в сумме 0, и *кубическим*, если во входных данных таких троек вообще нет.

Задачи с несколькими параметрами

Мы говорим об измерении производительности в виде функции *одного* параметра — обычно это значение аргумента из командной строки или объем входных данных. Но не так уж редко встречается и зависимость от нескольких параметров. Типичный пример — когда алгоритм вначале строит некую структуру данных, а затем выполняет последовательность операций с этой структурой данных. Параметрами такого приложения являются и размер структуры данных, и количество операций. Нам уже встречался такой пример — при анализе задачи белого списка с бинарным поиском, где берутся M чисел из белого списка и M чисел из входных данных; типичное время выполнения этой задачи пропорционально $M\lg N$.

Несмотря на все эти сложности, понимание сути порядка роста для времени выполнения отдельных программ ценно для любого программиста, а описанные здесь методы являются мощными и широко применимыми. Идея Кнута была в том, чтобы применить эти методы вплоть до мельчайших нюансов *в принципе* — и получать подробные и точные предсказания. Типичные вычислительные системы крайне сложны, и скрупулезный анализ лучше оставить экспертам, но подобные методы эффективны и для получения приблизительных оценок времени выполнении любой программы. Конструктору ракет достаточно знать, где завершится пробный полет: в океане или в городе; фармацевту необходимо знать, приведет ли заданный объем лекарства к смерти или выздоровлению подопытных; и любому ученому или инженеру, использующему компьютерную программу, необходимо хотя бы приблизительно представлять, отработает она за секунду или же за год.

Учет зависимости от входных данных

Для многих задач одной из наиболее существенных из перечисленных выше сложностей является зависимость от входных данных, т.е. существенный разброс значений времени выполнения. Время выполнения вышеописанной модификации `ThreeSum` может меняться от постоянного до кубического — в зависимости от входных данных — и для прогноза производительности необходим более тщательный анализ. Сейчас мы кратко рассмотрим ряд эффективных приемов, которые еще пригодятся при рассмотрении конкретных алгоритмов.

Модели входных данных

Один из способов — более тщательное моделирование видов входных данных, которые обрабатываются в рассматриваемых задачах.

Например, можно предположить, что числа, подаваемые на вход ThreeSum — случайные целые значения. Это предположение проблематично по двум причинам:

- модель может не соответствовать реальности;
- анализ может оказаться очень сложным и потребовать математических знаний, далеко выходящих за подготовку типичного студента или программиста.

Первая из этих причин более важна — зачастую потому, что целью вычислений как раз и является *выявление* характеристик входных данных. Например, если мы пишем программу обработки генома, то как оценить ее производительность для другого генома? Хорошая модель, описывающая геномы, которые встречаются в природе — это именно то, что пытаются найти ученые; поэтому оценка времени выполнения программ для данных, взятых из реальности, по сути, сводится к разработке такой модели! Вторая причина приводит к тому, что мы будем рассматривать математические результаты только для наиболее важных алгоритмов. Мы еще увидим несколько примеров, когда простая и удобная модель входных данных в сочетании с классическим математическим анализом помогает прогнозировать производительность.

Гарантии производительности в худшем случае

В некоторых приложениях необходимо, чтобы время выполнения программы было меньше определенной границы, независимо от входных данных. Для получения таких *гарантий* производительности теоретики придерживаются крайне пессимистического взгляда на скорость работы алгоритмов: каким будет время выполнения *в худшем случае*? Такой осторожный подход может быть, к примеру, уместен для программы, управляющей ядерным реактором, кардиостимулятором или тормозной системой автомобиля. В подобных случаях необходимо гарантировать, что такая программа завершит свою работу в пределах заданного времени, ведь иначе результат может оказаться катастрофическим.

При изучении реального мира ученые обычно не рассматривают худшие случаи: в биологии худшим случаем может оказаться исчезновение человеческой расы, а в физике — конец света. Но худший случай может оказаться очень реальным при разработке вычислительных систем, где входные данные могут быть сгенерированы другим пользователем (возможно, злоумышленником), а не природой. Например, веб-сайты, для которых не рассчитывалась гарантированная производительность, могут быть атакованы для получения *отказа в обслуживании* — это так называемая DoS-атака, когда хакеры заваливают сайт ненормальными запросами и заставляют его работать медленнее, чем планировалось. Поэтому многие наши алгоритмы спроектированы так, чтобы предоставить гарантии производительности, такие как описанные ниже.

Утверждение Г. В реализациях связными списками классов Bag (алгоритм 1.4), Stack (алгоритм 1.2) и Queue (алгоритм 1.3) все операции выполняются за постоянное время в худшем случае.

Доказательство. Следует непосредственно из кода. Количество инструкций, выполняемых для каждой операции, ограничено небольшой константой. *Внимание*: это рассуждение основано на (разумном) предположении, что система Java создает новый узел Node за постоянное время.

Рандомизированные алгоритмы

Один из важных способов обеспечить гарантированную производительность — введение случайности. Например, алгоритм быстрой сортировки, который мы будем изучать в разделе 2.3 (пожалуй, чаще всего применяемый алгоритм сортировки) в худшем случае выполняется за квадратичное время, однако случайное упорядочение входных данных дает вероятностную гарантию, что время выполнения будет линейно-логарифмическим. При каждом выполнении алгоритм работает разное время, но шанс, что это время не будет линейно-логарифмическим, настолько мал, что им можно пренебречь. Аналогично, алгоритмы хеширования для таблиц имен, которые будут рассматриваться в разделе 3.4 (также, пожалуй, наиболее распространенный способ), выполняются за линейное время в худшем случае, но за постоянное время с вероятностной гарантией. Эти гарантии не абсолютны, но вероятность того, что они не сработают, меньше шанса, что компьютер расплавится от удара молнии. Поэтому такие гарантии на практике не менее полезны, чем гарантии для худшего случая.

Последовательности операций

Во многих приложениях "входными данными" алгоритма могут быть не просто данные, а последовательность операций, выполненных клиентом. Например, стек, в который клиент сначала вталкивает N значений, а потом выталкивает их все, может весьма отличаться по времени выполнения от клиента, который выдает последовательность чередующихся N операций занесения и выборки. Наш анализ должен принимать во внимание обе такие ситуации (или ввести разумную модель последовательности операций).

Амортизационный анализ

Другой способ обеспечения гарантированной производительности — *амортизация* стоимости, когда вычисляется общая стоимость всех операций и делится на количество операций. При таком способе можно разрешить выполнение некоторых трудоемких операций; главное, чтобы не увеличилась средняя стоимость операций. Характерный пример подобного анализа — наше изучение изменения размера массива для структуры данных в классе `Stack`, которое было выполнено в разделе 1.3 (алгоритм 1.1 в листинге 1.3.5). Для простоты предположим, что N равно степени двух. Если начать с пустой структуры, то к скольким элементам массива выполняются обращения для N последовательных вызовов метода `push()`? Эту величину нетрудно посчитать: количество обращений к элементам массива равно

$$N + 4 + 8 + 16 + \ldots + 2N = 5N - 4$$

Первое слагаемое учитывает обращения к массиву в каждом из N вызовов метода `push()`, а последующие слагаемые — обращения к массиву для инициализации структуры данных при каждом удвоении ее размера. Поэтому *среднее количество обращений к массиву на одну операцию* постоянно, хотя последняя операция требует линейного времени. Этот прием называется "амортизационным" анализом потому, что стоимость нескольких трудоемких операций распределяется понемногу на большое количество несложных операций. На рис. 1.4.8 этот процесс проиллюстрирован с помощью класса `VisualAccumulator`.

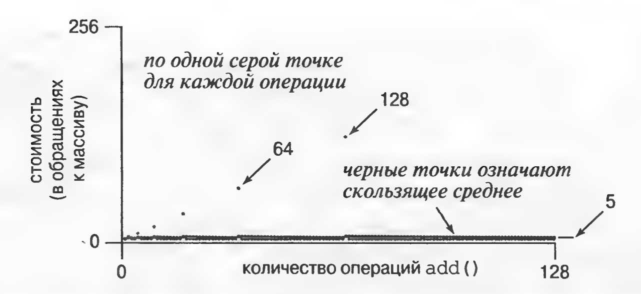


Рис. 1.4.8. Амортизированная стоимость добавления элемента в объект `RandomBag`

Утверждение Д. В реализации стека на основе массива переменного размера (алгоритм 1.1) среднее количество обращений к массиву для любой последовательности операций, начиная с пустой структуры данных, постоянно в худшем случае.

Набросок доказательства. Для каждой операции `push()`, которая приводит к увеличению массива (скажем, от размера N до размера $2N$), рассмотрим $N/2 + 2$ операций `push()`, которые в последнее время приводили к увеличению размера стека до k, для k от $N/2 + 2$ до N. Усреднение $4N$ обращений к массиву для его увеличения с $N/2$ обращениями к массиву (по одному для каждого вталкивания) дает среднее значение стоимости 9 обращений к массиву на операцию. Доказательство того, что количество обращений к массиву, используемое для любой последовательности M операций, пропорционально M, более сложно (см. упражнение 1.4.32).

Этот вид анализа применяется весьма широко. Между прочим, массивы с переменным размером мы еще будем использовать в качестве базовой структуры данных для некоторых алгоритмов, которые будут рассматриваться в данной книге.

Задача специалиста по алгоритмическому анализу — выявить как можно больше информации об алгоритме, а задача прикладного программиста — применить это знание для разработки программ, которые эффективно решают конкретные задачи. В идеале хотелось бы иметь алгоритмы, приводящие к ясному и лаконичному коду, который гарантирует хорошую производительность для актуальных данных. Многие классические алгоритмы, которые мы рассматриваем в данной главе, важны для очень многих приложений именно потому, что они обладают этими свойствами. На их основе можно самостоятельно разработать хорошие решения для типичных задач, которые встречаются в обычном программировании.

Память

Как и время выполнения, использование памяти программой непосредственно связано с физическим миром: значительный объем аппаратуры компьютера предназначен для хранения и выборки каких-то значений. Чем больше значений необходимо хранить в любой заданный момент времени, тем больше аппаратуры потребуется. Скорее всего, вы в курсе ограничений на память в вашем компьютере (больше, чем в отношении времени), поскольку, возможно, вы потратили деньги на приобретение дополнительной памяти.

Расход памяти четко определен для Java на любом конкретном компьютере (каждое значение требует в точности один и тот же объем памяти при каждом запуске программы), но Java реализована на очень многих вычислительных устройствах, и потребление памяти зависит от конкретных реализаций. Для экономии мы будем использовать слово *типичный*, чтобы указать, что значения зависят от особенностей машины.

Одна из самых значительных особенностей Java — ее система выделения памяти, которая предназначена для освобождения программиста от забот о памяти. Естественно, вы слышали или читали советы по использованию этой функции. Однако никто не снимает с вас обязанность знать — по крайней мере, приблизительно — когда требования программы к памяти могут помешать решению конкретной задачи.

Анализировать использование памяти гораздо проще, чем анализировать время выполнения — в основном потому, что при этом необходимо рассматривать не так много операторов программы (в основном объявления), а также потому, что анализ сложных объектов можно свести к примитивным типам, требования к памяти которых четко определены и понятны: достаточно сложить количество переменных с весами, соответствующими количеству байтов для каждого конкретного типа (табл. 1.4.9).

Например, тип данных `int` в Java представляет собой множество целых значений от –2 147 483 648 до 2 147 483 647 — всего это 2^{32} различных значений, и типичная Java-реализация использует 32 бита для представления значений типа `int`. Аналогичные соображения верны и для других примитивных типов: типичные реализации Java используют 8-битовые байты для хранения значений `char` в 2 байтах (16 битов), значений `int` в 4 байтах (32 бита), значений `double` и `long` в 8 байтах (64 бита) и значений `boolean` в 1 байте (поскольку компьютеры обычно обращаются к памяти побайтно). Зная объем доступной памяти, можно вычислить ограничения на эти значения. Например, при наличии на компьютере 1 Гбайт памяти (примерно 1 миллиард байтов) в нее может уместиться приблизительно 256 миллионов значений `int` или 128 миллионов значений `double`.

Таблица 1.4.9. Типичные требования к памяти для примитивных типов

Тип	Байтов
`boolean`	1
`byte`	1
`char`	2
`int`	4
`float`	4
`long`	8
`double`	8

С другой стороны, анализ использования памяти зависит от различий в оборудовании и реализациях Java, поэтому наши конкретные примеры следует рассматривать лишь в качестве указаний, каким образом можно определить требования к памяти, но не как окончательный диагноз для вашего компьютера. К примеру, во многих структурах используется некоторое представление машинных адресов, а объем памяти, необходимой для хранения машинного адреса, различается для различных машин. Для определенности мы будем считать, что для представления адресов необходимо 8 байтов — это типично для широко распространенных сейчас 64-разрядных архитектур — но все же помня, что во многих старых машинах используется 32-разрядная архитектура, где для машинного адреса требуется лишь 4 байта.

Объекты

Чтобы определить требования к памяти для некоторого объекта, понадобится сложить объемы памяти, используемые для каждой переменной экземпляров, и прибавить к сумме дополнительный объем, необходимый для заголовка каждого объекта — обычно это 16 байтов (рис. 1.4.9). В этот дополнительный объем входят ссылка на класс объекта, информация для сборки мусора и информация для синхронизации. Кроме того, необходимый объем памяти обычно дополняется до кратного 8 байтам (машинные слова на 64-разрядной машине). Например, объект `Integer` использует 24 байта: 16 байтов для заголовка объекта, 4 байта для переменной экземпляров типа `int` и еще 4 байта для дополнения до границы слова. Аналогично, объект `Date` (рис. 1.2.22) также использует 32 байта: 16 байтов для заголовка, по 4 байта для каждого из трех переменных экземпляров типа `int` и 4 байта на дополнение. Ссылка на объект, как правило, представляется адресом памяти, поэтому занимает 8 байтов. К примеру, объект `Counter` (рис. 1.2.21) использует 32 байта: 16 байтов на заголовок, 8 байтов для переменной типа `String` (ссылка), 4 байта для переменной типа `int` и 4 байта на дополнение. При расчете памяти на ссылку необходимо отдельно учитывать и память для самого указываемого объекта — в нашем случае память для значения `String` не учтена.

Связные списки

Вложенному не статическому (внутреннему) классу вроде нашего класса `Node` (см. подраздел "Связные списки" в разделе 1.3) требуется еще 8 дополнительных байтов — для ссылки на внешний экземпляр. Поэтому объект `Node` занимает 40 байтов: 16 байтов на заголовок объекта, по 8 байтов на ссылки на объекты `Item` и `Node` и еще 8 байтов на дополнительную ссылку. А поскольку объект `Integer` занимает 24 байта, стек из N целочисленных значений, созданный на основе представления связным списком (алгоритм 1.2), использует $32 + 64N$ байтов: 16 байтов на заголовок объекта `Stack`, 8 байтов на его переменную-ссылку, 4 байта для переменной типа `int` и 4 байта на дополнение до границы слова, а также по 64 байта на каждый элемент: 40 байтов для `Node` и 24 для `Integer`.

Массивы

Типичные требования к памяти для размещения различных видов массивов в Java показаны на рис. 1.4.10. Массивы в Java реализованы в виде объектов, обычно с дополнительным полем для хранения длины. *Массивы примитивных типов* обычно требуют 24 байта на служебную информацию (16 байтов на заголовок объекта, 4 байта на длину и 4 байта дополнения до границы слова) плюс память, необходимую для хранения значений. К примеру, массив N значений `int` занимает $24 + 4N$ байтов (с округлением до кратного 8), а массив N значений `double` — $24 + 8N$ байтов. *Массив объектов* представляет собой массив ссылок на объекты. Например, массив N объектов `Date` (рис. 1.2.22) занимает 24 байта (заголовок массива) плюс $8N$ байтов (ссылки) плюс 32 байта на каждый объект — итого $24 + 40N$ байтов. *Двумерный массив* реализуется как массив массивов (а каждый массив — как объект).

К примеру, двумерный массив размером $M \times N$ значений `double` занимает 24 байта (заголовок массива массивов) плюс $8M$ байтов (ссылки на массивы строк) плюс M раз по 24 байта (заголовки массивов строк) плюс M раз по N раз по 8 байтов (для N значений `double` в каждой из M строк) — итого $8NM + 32M + 24 \sim 8NM$ байтов. Если элементами массива являются объекты, то аналогичный подсчет дает $8NM + 32M + 24 \sim 8NM$ байтов для массива массивов ссылок на объекты плюс память для самих объектов.

7 Зак. 3239

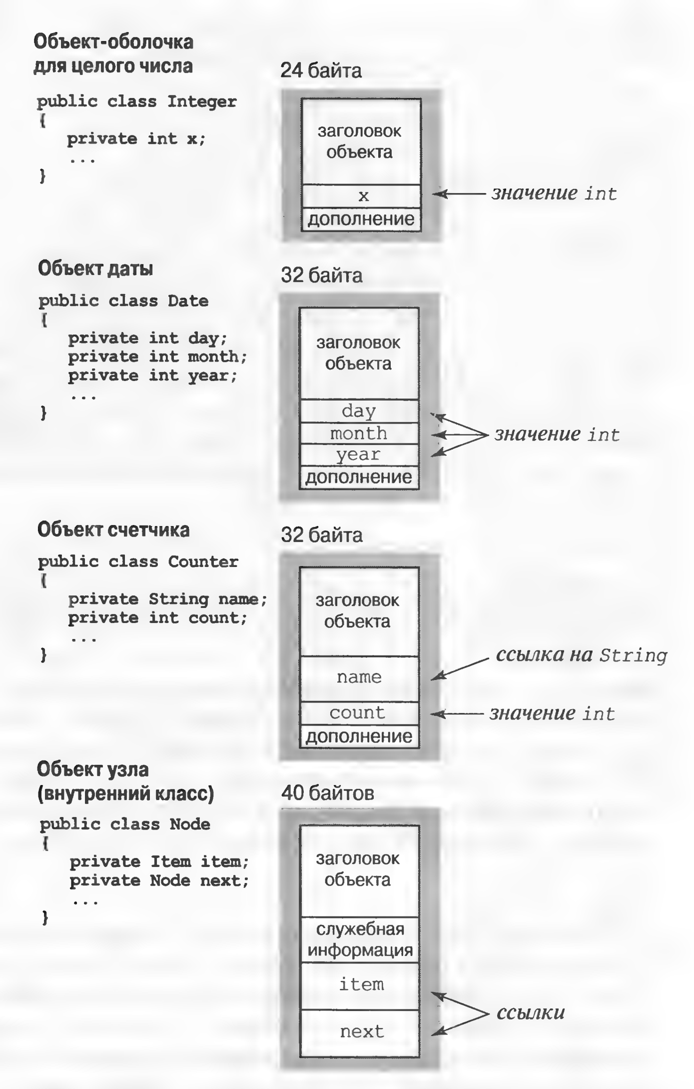
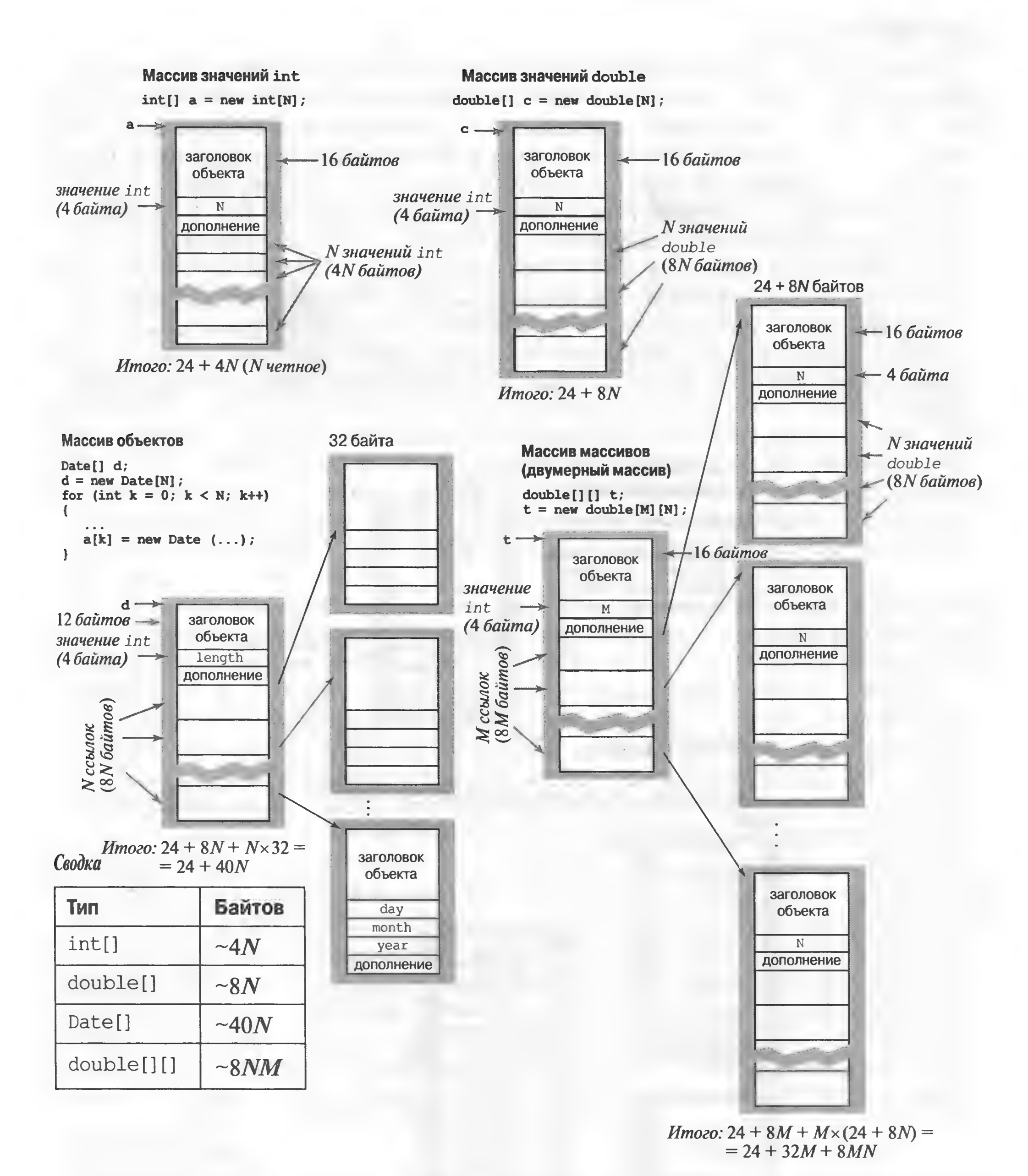


Рис. 1.4.9. Типичные требования к памяти для объектов



Тип	Байтов
int[]	~4N
double[]	~8N
Date[]	~40N
double[][]	~8NM

Рис. 1.4.10. Типичные требования к памяти для массивов значений `int`*, значений* `double`*, объектов и массивов*

Строковые объекты

Объем памяти для Java-объектов `String` подсчитывается так же, как и для любого другого объекта, только необходимо учитывать распространенное в случае строк создание псевдонимов. Стандартная реализация `String` содержит четыре переменных экземпляра: ссылку на массив символов (8 байтов) и три значения `int` (по 4 байта). Первое значение `int` представляет собой смещение в символьном массиве, второе — счетчик (длина строки). С учетом имен переменных, приведенных на рис. 1.4.11, представленная там строка состоит из символов с `value[offset]` по `value[offset + count - 1]`. Третье значение `int` в объектах `String` содержит хеш-код — он облегчает вычисления в ряде случаев, которые сейчас рассматриваться не будут. Следовательно, каждый объект `String` занимает в общей сложности 40 байтов: 16 байтов на заголовок объекта плюс 4 байта на каждую из трех переменных `int` плюс 8 байтов на ссылку на массив плюс 4 байта на выравнивание. И это без учета памяти под сами символы, находящиеся в массиве.

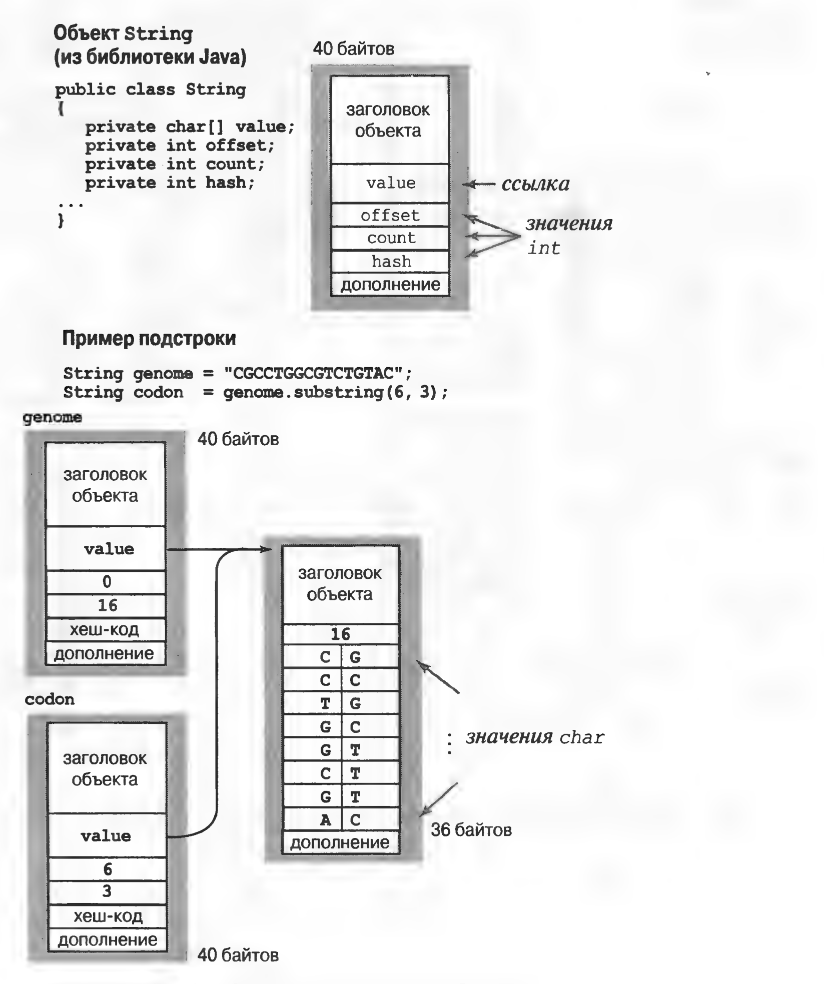


Рис. 1.4.11. Объект `String` и подстрока

Память, необходимая для хранения символов, считается отдельно, т.к. строки часто совместно используют один массив `char`. Поскольку объекты `String` неизменяемые, это помогает экономить память, когда объекты `String` используют один и тот же массив `value[]`.

Значения строк и подстроки

Строка длиной N обычно занимает 40 байтов (для объекта `String`) плюс $24 + 2N$ байтов (для массива, содержащего символы) — итого $64 + 2N$ байтов. Но при обработке строк часто формируются подстроки, и представление строк в Java предназначено для того, чтобы работать без копирования символов строк. Метод `substring()` создает новый объект `String` (40 байтов), но использует тот же массив `value[]`, поэтому подстрока существующей строки занимает только 40 байтов. В объекте для подстроки создается псевдоним в виде символьного массива, содержащего исходную строку; эта подстрока определяется полями смещения и длины. То есть *подстрока занимает постоянный объем дополнительной памяти, и формирование подстроки требует постоянного времени*, даже при очень больших длинах строки и подстроки. Примитивное представление, требующее копирования символов для создания подстрок, потребовало бы линейного времени и объема памяти. Возможность создания подстроки с затратами памяти и времени, не зависящими от ее длины — ключ к эффективности во многих базовых алгоритмах обработки строк.

Эти базовые механизмы эффективны для оценки потребности в памяти очень многих программ, но имеется много факторов, которые усложняют задачу. Мы уже сказали о возможном эффекте создания псевдонима. Более того, потребность в памяти, если задействованы вызовы функций — это сложный динамический процесс, т.к. при этом важную роль играет системный механизм выделения памяти со многими зависимостями. Например, если программа вызывает метод, система выделяет память, необходимую этому методу (для его локальных переменных) из специальной области памяти, которая называется *стеком*, а когда метод возвращает управление вызвавшей его программе, эта память возвращается в стек. По этой причине опасно создавать массивы или другие большие объекты в рекурсивных программах, поскольку каждый рекурсивный вызов связан со значительным расходом памяти. При создании объекта с помощью операции `new` система выделяет память, необходимую для объекта, из другой специальной области памяти, которая называется *кучей* (это не то же самое, что и бинарное сортирующее дерево, которое иногда называют кучей и которое будет рассматриваться в разделе 2.4). Необходимо помнить, что каждый объект существует до того момента, когда на него уже не указывает ни одна ссылка, и тогда занимаемая им память возвращается в кучу с помощью системного процесса, называемого *сборкой мусора*. Такие динамические моменты могут значительно усложнить задачу точной оценки потребности программы в памяти.

Перспектива

Хорошая производительность всегда важна. Очень медленно работающая программа почти так же бесполезна, как и функционирующая с ошибками — поэтому, конечно, следует с самого начала уделять серьезное внимание трудоемкости, чтобы иметь некоторое представление, какие задачи будут разрешимы, а какие нет. В частности, рекомендуется всегда хорошо представлять себе, что за код составляет внутренний цикл программы.

Пожалуй, вторая по распространенности ошибка в программировании — игнорирование характеристик производительности. Более быстрые алгоритмы обычно более

сложны, чем примитивные, поэтому возникает желание использовать медленный алгоритм, чтобы не возиться со сложным кодом. Но иногда можно достичь серьезной экономии с помощью всего лишь нескольких строк хорошо продуманного кода. Пользователи удивительно большого количества вычислительных систем тратят значительное время, ожидая, когда примитивные квадратичные алгоритмы завершат решение задачи, хотя доступны линейные или линейно-логарифмические алгоритмы, которые могут решить те же задачи за доли секунды. А при работе с задачами большого размера у нас вообще нет выбора, и приходится искать лучшие алгоритмы.

Обычно мы будем по умолчанию применять описанную в данном разделе методологию, чтобы оценивать потребность в памяти и разрабатывать гипотезы о порядке роста для времени выполнения на основе аппроксимации старшим членом, которая следует из математического анализа модели стоимости, а затем экспериментально проверять эти гипотезы. Усовершенствование программы, которое делает ее более ясной, эффективной и элегантной, должно быть вашей целью во время работы над этой программой. Если при разработке программы постоянно помнить о ее трудоемкости, то при каждом ее выполнении вы будете с благодарностью вспоминать об этом.

Вопросы и ответы

Вопрос. **Почему бы не подключить библиотеку `StdRandom` вместо использования файла `1Mints.txt`?**

Ответ. Так легче отлаживать код при разработке и при воспроизведении экспериментов. Библиотека `StdRandom` выдает при каждом вызове разные значения, поэтому запуск программы после коррекции ошибки может не проверить работу исправления! Для устранения этой проблемы можно воспользоваться методом `initialize()` из `StdRandom`, но эталонный файл вроде `1Mints.txt` облегчает добавление тестовых вариантов при отладке. Кроме того, различные программисты могут сравнивать производительность на различных компьютерах, не заботясь о синхронизации модели входных данных. Но после завершения отладки программы и получения четкого представления о ее производительности вполне разумно протестировать ее и на случайных данных. Такой подход принят в программах `DoublingTest` и `DoublingRatio`.

Вопрос. **Я запускал программу `DoublingRatio` на своем компьютере, но результаты не такие последовательные, как в книге. Некоторые отношения далеки от 8. Почему?**

Ответ. Об этом мы говорили в подразделе “Предостережения”. Скорее всего, операционная система вашего компьютера решила сделать во время эксперимента что-то еще. Один из способов устранения подобных проблем — потратить больше времени на дополнительные эксперименты. Например, программу `DoublingTest` можно изменить так, чтобы выполнять по 1000 экспериментов для каждого N и, таким образом, получить гораздо более точные оценки времени выполнения для каждого размера (см. упражнение 1.4.39).

Вопрос. **Что в точности означает фраза “при увеличении N” в определении аппроксимации старшим членом?**

Ответ. Формальное определение отношения $f(N) \sim g(N)$ выглядит так:

$$\lim_{N\to\infty} f(N)/g(N) = 1$$

Вопрос. **Я встречал другие обозначения для описания порядка роста. В чем дело?**

Ответ. Широко используется обозначение с большой буквой O: говорят, что $f(N)$ имеет порядок $O(g(N))$, если существуют константы c и N, такие, что $|f(N)| < cg(N)$ для всех $N > N_0$. Эта нотация очень удобна для получения асимптотических верхних границ производительности алгоритмов, что важно в теории алгоритмов. Но она бесполезна для прогнозирования производительности или для сравнения алгоритмов.

Вопрос. **А почему бесполезна?**

Ответ. В основном потому, что она описывает лишь *верхнюю границу* времени выполнения. Реальная производительность может оказаться гораздо лучше. Время выполнения алгоритма может быть и $O(N^2)$, и $\sim aN\log N$. Поэтому такую нотацию нельзя применять для выполнения проверок вроде нашего теста отношения с удвоением (см. утверждение В).

Вопрос. **Тогда почему *O*-обозначение так широко используется?**

Ответ. Оно облегчает вывод граничных значений для порядка роста, даже для сложных алгоритмов, для которых более точный анализ может оказаться невозможным. Более того, оно совместимо с Ω- и Θ-нотациями, которые применяют теоретики для классификации алгоритмов по границам их производительности в худшем случае. Говорят, что $f(N)$ имеет порядок $\Omega(g(N))$, если существуют такие константы c и N_0, что $|f(N)| > cg(N)$ для $N > N_0$; а если $f(N)$ имеет порядок $O(g(N))$ и $\Omega(g(N))$, то говорят, что $f(N)$ имеет порядок $\Theta(g(N))$. Ω-нотация обычно применяется для описания *нижней границы* в худшем случае, а Θ-нотация — для описания производительности алгоритмов, которые *оптимальны* в том смысле, что никакой алгоритм не может иметь лучший асимптотический порядок роста в худшем случае. Оптимальные алгоритмы, несомненно, стоит рассматривать в связи с практическими приложениями, но, как мы увидим, важны и другие соображения.

Вопрос. **А разве не важны верхние границы асимптотической производительности?**

Ответ. Важны, но мы предпочитаем рассматривать точные результаты, зависящие от частоты выполнения операторов и моделей стоимости, потому что они дают больше информации о производительности алгоритма и потому что получение таких результатов возможно для рассматриваемых нами алгоритмов. Например, мы говорим: “программа `ThreeSum` выполняет $\sim N^3/2$ обращений к массиву” или “количество выполнений оператора `cnt++` в программе `ThreeSum` равно $\sim N^3/6$ в худшем случае”. Это несколько более пространно, но зато гораздо информативнее заявления “время выполнения `ThreeSum` равно $O(N^3)$”.

Вопрос. **Если порядок роста для времени выполнения алгоритма равен $N\log N$, то тест с удвоением приведет к гипотезе, что время выполнения равно $\sim aN$ для некоторой константы a. Но ведь это ошибка?**

Ответ. Лучше не делать выводы о конкретной математической модели на основе только экспериментальных данных, однако если мы просто пытаемся спрогнозировать производительность, это совсем не проблема. Например, для N между 16 000 и 32 000 графики $14N$ и $N\lg N$ очень близки друг к другу. Данные хорошо аппроксимируются обеими кривыми. При увеличении N эти кривые становятся все ближе. Для экспериментальной проверки гипотезы, что время выполнения является линейно-логарифмическим, а не линейным, потребуется тщательное тестирование.

Вопрос. **Можно ли считать, что оператор `int[] a = new int[N]` выполняет N обращений к массиву (для инициализации всех элементов нулями)?**

Ответ. Скорее всего да, поэтому мы предполагаем в настоящей книге, что это именно так, хотя хитроумный компилятор может попытаться избежать этих затрат для больших разреженных массивов.

Упражнения

1.4.1. Покажите, что количество различных троек, которые можно выбрать из N элементов, равно $N(N-1)(N-2)/6$. *Совет*: воспользуйтесь математической индукцией.

1.4.2. Измените программу `ThreeSum`, чтобы она корректно работала для больших значений `int` — даже таких, что сложение двух их них может привести к переполнению.

1.4.3. Измените программу `DoublingTest`, чтобы она с помощью класса `StdDraw` формировала графики с обычными и логарифмическими осями (как в тексте), при необходимости выполняя масштабирование, чтобы график всегда занимал существенную часть окна.

1.4.4. Составьте таблицу наподобие табл. 1.4.3 для программы `TwoSum`.

1.4.5. Приведите аппроксимации ведущими членами для следующих зависимостей:

а) $N + 1$

б) $1 + 1/N$

в) $(1 + 1/N)(1 + 2/N)$

г) $2N^3 - 15N^2 + N$

д) $\lg(2N) / \lg N$

е) $\lg(N^2 + 1) / \lg N$

ж) $N^{100} / 2^N$

1.4.6. Приведите порядок роста (в виде функции от N) для времени выполнения каждого из следующих кодовых фрагментов:

```
int sum = 0;
for (int n = N; n > 0; n /= 2)
   for(int i = 0; i < n; i++)
      sum++;

int sum = 0;
for (int i = 1 i < N; i *= 2)
   for (int j = 0; j < i; j++)
      sum++;

int sum = 0;
for (int i = 1 i < N; i *= 2)
   for (int j = 0; j < N; j++)
      sum++;
```

1.4.7. Проанализируйте производительность программы `ThreeSum` с моделью стоимости, которая учитывает арифметические операции (и сравнения) с вводимыми числами.

1.4.8. Напишите программу для определения количества пар значений из входного файла, равных друг другу. Если ваша первая попытка будет квадратичной, подумайте еще раз и воспользуйтесь методом `Arrays.sort()`, чтобы получить линейно-логарифмическое решение.

1.4.9. Приведите формулу для прогноза времени выполнения программы для решения задачи размером N, если эксперименты с удвоением размера показали, что коэффициент удвоения равен 2^b, а время выполнения для задач с размером N_0 равно T.

1.4.10. Измените алгоритм бинарного поиска, чтобы он всегда возвращал элемент с наименьшим индексом, который равен искомому элементу (и все-таки гарантировал логарифмическое время выполнения).

1.4.11. Добавьте в класс `StaticSETofInts` (рис. 1.2.26) метод `howMany()`, который находит количество вхождений заданного ключа за время, пропорциональное $\log N$ в худшем случае.

1.4.12. Напишите программу, которая для двух сортированных массивов N значений `int` выводит упорядоченный список всех элементов, которые присутствуют в обоих массивах. Время выполнения программы должно быть пропорционально N в худшем случае.

1.4.13. Используя предположения из текста, приведите объемы памяти, необходимые для представления объектов каждого из следующих типов:

а) `Accumulator`

б) `Transaction`

в) `FixedCapacityStackOfStrings` с емкостью C и N элементами

г) `Point2D`

д) `Interval1D`

е) `Interval2D`

ж) `Double`

Творческие задачи

1.4.14. *Четверки*. Разработайте алгоритм для задачи *поиска четверок с нулевой суммой*.

1.4.15. *Быстрые тройки*. В качестве разминки разработайте реализацию `TwoSumFaster`, в которой используется *линейный* алгоритм подсчета пар с нулевой суммой после сортировки этого массива (вместо линейно-логарифмического алгоритма на основе бинарного поиска). Затем примените аналогичный принцип для разработки квадратичного алгоритма для задачи поиска троек с нулевой суммой.

1.4.16. *Ближайшая пара (одномерный случай)*. Напишите программу, которая находит в заданном массиве `a[]` из N значений `double` *ближайшую пару* — т.е. два значения, разность которых не больше разности для любой другой пары (по абсолютной величине). Время выполнения программы должно быть линейно-логарифмическим в худшем случае.

1.4.17. *Самая далекая пара (одномерный случай)*. Напишите программу, которая находит в заданном массиве `a[]` из N значений `double` *самую далекую пару* — т.е. два значения, разность которых не меньше разности для любой другой пары (по аб-

солютной величине). Время выполнения программы должно быть линейным в худшем случае.

1.4.18. *Локальный минимум в массиве.* Напишите программу, которая находит в заданном массиве `a[]` из N различных целочисленных значений *локальный минимум* — такой индекс `i`, что `a[i] < a[i-1]` и `a[i] < a[i+1]`. Программа должна использовать ~$2\lg N$ сравнений в худшем случае.

Ответ. Сравните среднее значение `a[N/2]` с его соседями `a[N/2 - 1]` и `a[N/2 + 1]`. Если `a[N/2]` является локальным минимумом, то программа заканчивает работу, иначе переходит к поиску в половине с меньшим соседом.

1.4.19. *Локальный минимум в матрице.* Пусть дан массив `a[]` размером $N \times N$, содержащий N^2 различных целых чисел. Разработайте алгоритм со временем выполнения, пропорциональным N, который находит *локальный минимум* — такую пару индексов `i` и `j`, что `a[i][j] < a[i+1][j]`, `a[i][j] < a[i][j+1]`, `a[i][j] < a[i-1][j]` и `a[i][j] < a[i][j-1]`. Время выполнения алгоритма должно быть пропорционально N в худшем случае.

1.4.20. *Битонический поиск.* Массив называется *битоническим*, если он состоит из возрастающей последовательности целых чисел, за которой идет убывающая последовательность целых чисел. Напишите программу, которая при заданном битоническом массиве N целых чисел определяет, присутствует ли в массиве заданное число. Эта программа должна использовать ~$3\lg N$ сравнений в худшем случае.

1.4.21. *Бинарный поиск по различным значениям.* Разработайте реализацию бинарного поиска для АТД `StaticSETofInts` (см. рис. 1.2.26), в которой время выполнения метода `contains()` гарантированно равно ~$\lg R$, где R — количество различных целых чисел в массиве, передаваемое конструктору в качестве аргумента.

1.4.22. *Бинарный поиск только со сложением и вычитанием* [Михай Патраску]. Напишите программу, которая для заданного массива из N различных значений `int`, упорядоченных по возрастанию, определяет, присутствует ли в нем заданное целое число. Разрешается использовать только операции сложения и вычитания и постоянный объем дополнительной памяти. Время выполнения программы должно быть пропорционально $\log N$ в худшем случае.

Ответ. Вместо поиска на основе степеней двойки (бинарный поиск) используйте числа Фибоначчи (которые также возрастают экспоненциально). Поддерживайте текущий интервал поиска в виде $[i, i + F_k]$ и храните в двух переменных значения F_k и F_{k-1}. На каждом шаге вычисляйте F_{k-2} с помощью вычитания, проверяйте элемент $i + F_{k-2}$ и заменяйте текущий интервал либо $[i, i + F_{k-2}]$, либо $[i + F_{k-2}, i + F_{k-2} + F_{k-1}]$.

1.4.23. *Бинарный поиск дроби.* Разработайте метод, который использует логарифмическое количество запросов вида "Меньше ли число, чем x?" для поиска такого рационального числа p/q, что $0 < p < q < N$. *Подсказка*: две дроби со знаменателями, меньшими N, не могут отличаться более чем на $1/N^2$.

1.4.24. *Выбрасывание яиц из здания.* Пусть имеется N-этажное здание и много яиц. Пусть также яйцо разбивается, если оно выбрасывается с этажа F или выше. Вначале разработайте стратегию для определения значения F, разбив ~$\lg N$ яиц и выбросив ~$\lg N$, а затем найдите способ снизить стоимость до ~$2\lg F$.

1.4.25. *Выбрасывание двух яиц из здания.* Рассмотрим предыдущую задачу, но только теперь у нас лишь два яйца, а модель стоимости — количество бросаний. Разработайте такую стратегию для определения F, чтобы количество бросаний не превышало $2N$, а затем найдите способ снизить стоимость до $\sim cF$. Это аналогично ситуации, когда попадания при поиске (целые яйца) обходятся гораздо дешевле неудач (разбитые яйца).

1.4.26. *Коллинеарные тройки.* Допустим, имеется алгоритм, который принимает в качестве входных данных N различных точек на плоскости и возвращает количество троек точек, которые лежат на одной прямой. Покажите, что этот алгоритм можно использовать для решения задачи поиска троек с нулевой суммой. *Совет*: докажите алгебраически, что точки (a, a^3), (b, b^3) и (c, c^3) коллинеарны тогда и только тогда, когда $a + b + c = 0$.

1.4.27. *Очередь из двух стеков.* Реализуйте очередь на основе двух стеков, чтобы каждая операция с очередью требовала амортизировано постоянного количества операций со стеками. *Подсказка*: если затолкнуть элементы в стек, а потом вытолкнуть их все, их порядок изменится на обратный. Повторение этого процесса возвратит исходный порядок.

1.4.28. *Стек из очереди.* Реализуйте стек на основе одной очереди, чтобы все операции со стеком выполнялись за линейное количество операций с очередью. *Подсказка*: для удаления элемента нужно извлечь из очереди все элементы по одному и поместить их в конец очереди — кроме последнего, который нужно удалить и возвратить. (Конечно, такое решение крайне неэффективно.)

1.4.29. *Стеко-очередь из двух стеков.* Реализуйте стеко-очередь на основе двух стеков, чтобы каждая операция со стеко-очередью (см. упражнение 1.3.32) требовала амортизировано постоянного количества операций со стеками.

1.4.30. *Дек из стека и стеко-очереди.* Реализуйте дек на основе стека и стеко-очереди (см. упражнение 1.3.32), чтобы каждая операция с деком требовала амортизировано постоянного количества операций со стеком и стеко-очередью.

1.4.31. *Дек из трех стеков.* Реализуйте дек на основе трех стеков, чтобы каждая операция с деком требовала амортизировано постоянного количества операций со стеками.

1.4.32. *Амортизационный анализ.* Докажите, что если начать с пустого стека, количество обращений к массиву, выполняемых при любой последовательности M операций в реализации класса `Stack` на основе массива с переменным размером, пропорционально M.

1.4.33. *Требования к памяти на 32-разрядной машине.* Приведите требования к памяти для типов `Integer`, `Date`, `Counter`, `int[]`, `double[]`, `double[][]`, `String`, `Node` и `Stack` (на основе связного списка) для 32-разрядной машины. Считайте, что ссылка занимает 4 байта, заголовок объекта — 8 байтов, а выравнивание выполняется до границы 4 байтов.

1.4.34. *"Горячо–холодно".* Ваша цель — угадать секретное число от 1 до N. Вы предлагаете целые числа от 1 до N. После каждой попытки вы узнаете, угадали ли вы число (и игра прекращается). Иначе вам сообщается, "теплее" (ближе) или "холоднее" (дальше) предложенное вами число от предыдущего. Разработайте алгоритм, который находит секретное число за не более чем $\sim 2\lg N$ попыток. Затем разработайте другой алгоритм, который находит секретное число за не более чем $\sim 1\lg N$ попыток.

1.4.35. *Затраты времени для стеков.* Проверьте правильность данных, приведенных в табл. 1.4.10 — типичные трудоемкости для различных реализаций стека на основе модели стоимости, в которой учитываются и *обращения к данным* (обращения к данным, которые занесены в стек, обращения к элементам массива или обращения к переменным экземпляра объекта), и *создания объектов*.

Таблица 1.4.10. Временная сложность для стеков (различные реализации)

Структура данных	Тип элемента	Стоимость вталкивания N значений `int`	
		Обращения к данным	Создание объектов
Связный список	`int` `Integer`	$2N$ $3N$	$2N$ $2N$
Массив переменного размера	`int` `Integer`	$\sim 5N$ $\sim 5N$	$\lg N$ $\sim N$

1.4.36. *Затраты памяти для стеков.* Проверьте правильность данных, приведенных в табл. 1.4.11 — типичные затраты памяти для различных реализаций стека. Используйте статический вложенный класс для узлов связного списка и заголовок для нестатического вложенного класса.

Таблица 1.4.11. Требования к памяти для стеков (различные реализации)

Структура данных	Тип элемента	Затраты памяти на N значений `int` (в байтах)
Связный список	`int` `Integer`	$\sim 32N$ $\sim 56N$
Массив переменного размера	`int` `Integer`	от $\sim 4N$ до $\sim 16N$ от $\sim 32N$ до $\sim 56N$

Эксперименты

1.4.37. *Снижение производительности из-за автоупаковки.* Экспериментально определите снижение производительности на вашем компьютере из-за использования автоупаковки и автораспаковки. Разработайте реализацию `FixedCapacityStackOfInts` и используйте клиент наподобие `DoublingRatio`, чтобы сравнить ее производительность с обобщенным типом `FixedCapacityStack<Integer>` для большого количества операций `push()` и `pop()`.

1.4.38. *Наивная реализация для нулевых троек.* Экспериментально оцените следующую реализацию внутреннего цикла в программе `ThreeSum`:

```
for (int i = 0; i < N; i++)
   for (int j = 0; j < N; j++)
      for (int k = 0; k < N; k++)
         if (i < j && j < k)
            if (a[i] + a[j] + a[k] == 0)
               cnt++;
```

Для этого разработайте версию `DoublingTest`, которая вычисляет отношение времени выполнения этой программы и `ThreeSum`.

1.4.39. *Повышенная точность для теста с удвоением.* Измените программу DoublingRatio, чтобы она принимала из командной строки второй аргумент, который задает количество вызовов timeTrial() для каждого значения N. Выполните полученную программу для 10, 100 и 1000 попыток и прокомментируйте точность полученных результатов.

1.4.40. *Тройки с нулевой суммой для случайных значений.* Сформулируйте и проверьте гипотезу относительно количества троек из N случайных значений int, сумма которых равна 0. При наличии хорошей математической подготовки разработайте подходящую математическую модель для этой задачи, если значения равномерно распределены между $-M$ и M, где M — большое значение.

1.4.41. *Значения времени выполнения.* Оцените время, необходимое для выполнения на вашем компьютере программ TwoSumFast, TwoSum, ThreeSumFast и ThreeSum для решения задач с файлом, хранящим 1 миллион чисел. Воспользуйтесь для этого программой DoublingRatio.

1.4.42. *Размеры задач.* Определите максимальное значение P, для которого еще можно выполнить на вашем компьютере программы TwoSumFast, TwoSum, ThreeSumFast и ThreeSum для решения задач с файлом, хранящим 2^P тысяч чисел. Воспользуйтесь для этого программой DoublingRatio.

1.4.43. *Сравнение массивов переменного размера и связных списков.* Экспериментально проверьте гипотезу, что стеки на основе массивов с переменным размером работают быстрее, чем стеки на основе связных списков (см. упражнения 1.4.35 и 1.4.36). Для этого разработайте версию программы DoublingRatio, которая вычисляет отношение значений времени выполнения для двух программ.

1.4.44. *Задача дня рождения.* Напишите программу, которая принимает из командной строки целое число N и с помощью метода StdRandom.uniform() генерирует случайную последовательность целых чисел из диапазона между 0 и $N-1$. Экспериментально проверьте гипотезу, что количество чисел, сгенерированных до первого повторяющегося значения, равно $\sim\sqrt{\pi N/2}$.

1.4.45. *Задача инкассатора.* Генерируйте случайные целые числа, как в предыдущем упражнении, и экспериментально проверьте гипотезу, что количество чисел, сгенерированных до получения всех возможных значений, равно $\sim N H_N$.

1.5. Учебный пример: объединение–сортировка

Для демонстрации нашего базового подхода к разработке и анализу алгоритмов мы рассмотрим подробный пример. С его помощью будут освещены следующие темы.

- Хорошие алгоритмы демонстрируют (и формируют) различие между возможностью решить практическую задачу и невозможностью даже подступиться к ней.
- Эффективный алгоритм можно закодировать не сложнее, чем неэффективный.
- Выявление характеристик производительности конкретной реализации может оказаться интересным и полезным интеллектуальным занятием.
- Научный метод — важный инструмент выбора из различных методов для решения одной и той же задачи.

В настоящей книге мы будем еще не раз касаться всех этих тем. Рассматриваемый характерный пример закладывает фундамент нашего использования общей методологии для многих других задач.

Проблема, которую мы будем сейчас рассматривать, далеко не игрушечная: это фундаментальная вычислительная задача, и решение, которое мы разработаем, может быть полезно во множестве приложений — от просачивания в физической химии до связности в электронных сетях. Мы начнем с простого решения, а затем постараемся оценить характеристики производительности этого решения, что поможет понять способы усовершенствования алгоритма.

Динамическая связность

Мы начнем со следующей формулировки задачи. Имеется последовательность вводимых пар целых чисел, где каждое число представляет объект некоторого типа, и пара `p q` означает, что "`p` связан с `q`". Мы считаем, что отношение "связан с" является отношением *эквивалентности*, т.е. что оно

- *рефлексивно* — `p` связан с `p`;
- *симметрично* — если `p` связан с `q`, то и `q` связан с `p`;
- *транзитивно* — если `p` связан с `q` и `q` связан с `r`, то `p` связан с `r`.

Отношение эквивалентности разбивает объекты на *классы эквивалентности*. В этом случае два объекта принадлежат одному классу эквивалентности тогда и только тогда, когда они связаны. Нам нужно написать программу, которая отсеивает из последовательности лишние пары (в которых оба объекта принадлежат одному и тому же классу эквивалентности). То есть после чтения пары `p q` из входных данных программа должна вывести эту пару в выходные данные только в том случае, если из пар, просмотренных до нее, *не следует*, что `p` и `q` связаны. Если из предыдущих пар *следует*, что `p` связан с `q`, программа должна проигнорировать пару `p q` и перейти к чтению следующей пары. Пример такого процесса приведен на рис. 1.5.1.

Для достижения этой цели необходима структура данных, которая может запоминать информацию о просмотренных парах, позволяющую определить, связана ли новая пара объектов. Неформально задача создания такого метода называется задачей *динамической связности*. Она возникает в перечисленных ниже областях.

Сети

Целые числа могут представлять компьютеры в большой сети, а пары — связи в этой сети. В этом случае наша программа определяет, нужно ли создавать новое прямое соединение, чтобы p и q могли обмениваться данными, или же можно проложить путь через существующие соединения. Целые числа могут также представлять контакты в электронной схеме, а пары — проводники, соединяющие эти контакты. Либо это могут быть люди в социальной сети, а пары — дружеские отношения между ними. В таких случаях может понадобиться обрабатывать миллионы объектов и миллиарды связей.

Эквивалентность имен переменных

В некоторых средах программирования можно объявлять, что два имени переменных эквивалентны (указывают на один и тот же объект). После серии таких объявлений системе необходимо определять, эквивалентны ли два заданных имени. Это потребность возникла давно (для языка программирования FORTRAN) и как раз была одной из причин разработки алгоритмов, о которых пойдет речь в данном разделе.

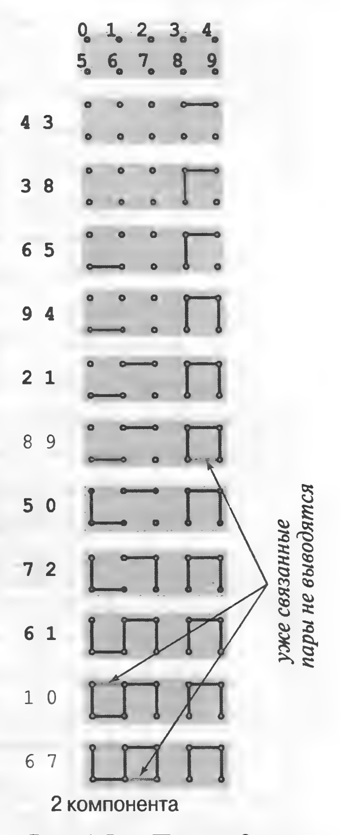


Рис. 1.5.1. *Пример динамической связности*

Математические множества

На более абстрактном уровне целые числа можно считать принадлежащими различным математическим множествам. При обработке пары p q спрашивается, принадлежат ли числа p и q одному множеству. Если нет, множества, содержащие p и q, объединяются, и после этого данные числа уже принадлежат одному множеству.

Для определенности до конца этого раздела мы будем использовать сетевую терминологию и называть объекты *узлами*, пары — *соединениями*, а классы эквивалентности — *связными компонентами* или, для краткости, просто *компонентами*. Для простоты будем предполагать, что имеется N узлов с целочисленными именами от 0 до N-1. Потери общности при этом не происходит, т.к. в главе 3 будет рассмотрено много алгоритмов, которые могут эффективно связывать произвольные имена с такими целыми идентификаторами.

Пример большего размера, который может дать некоторое представление о сложности задачи связности, приведен на рис. 1.5.2. Нетрудно увидеть компонент, состоящий из единственного узла в середине левой части диаграммы, и компонент из пяти узлов внизу слева, но весьма сложно проверить, что все другие узлы соединены между собой. А ведь программе еще труднее, чем нам: ей приходится работать только с именами узлов и соединениями, у нее нет доступа к геометрическому изображению узлов на диаграмме. Как можно быстро определить, соединены ли два заданных узла?

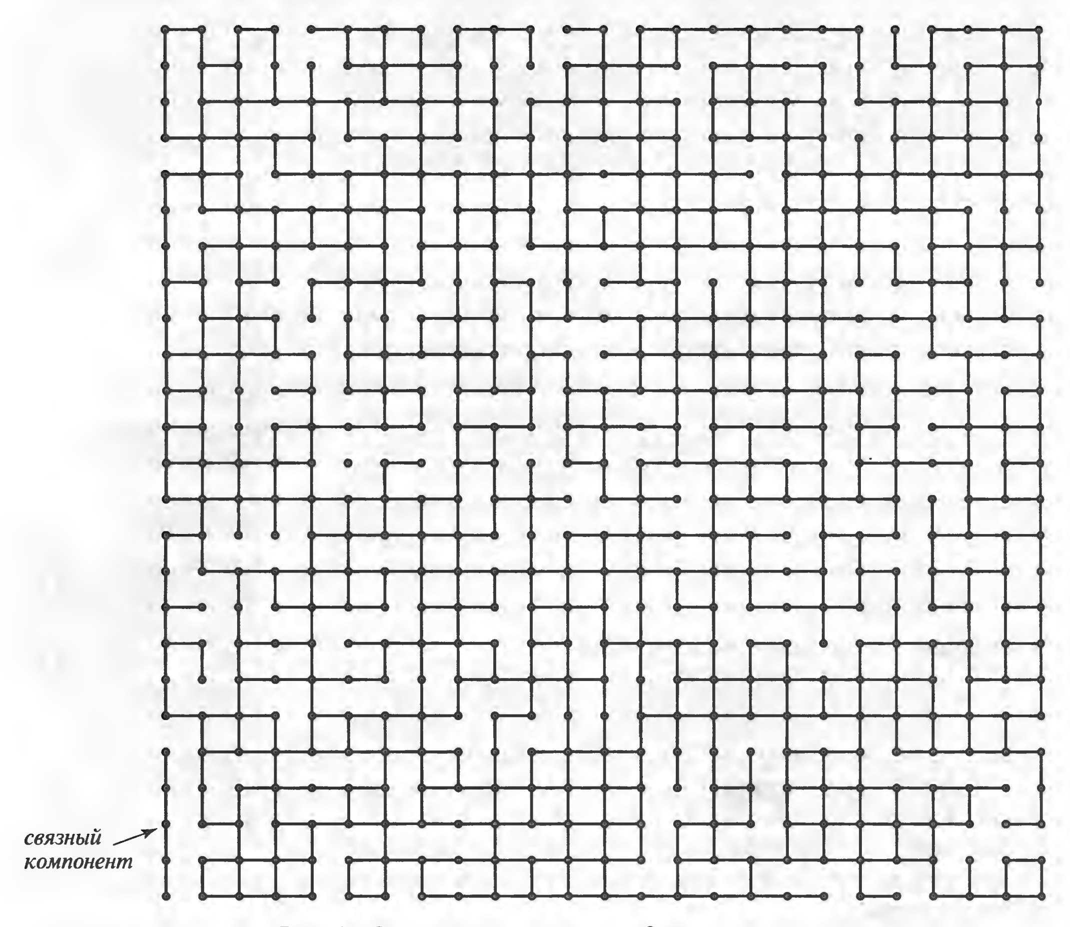


Рис. 1.5.2. Пример связности среднего размера (625 узлов, 900 ребер, 3 связных компонента)

Первое, что нам необходимо сделать при разработке алгоритма — точно сформулировать задачу. Чем больше наши требования к алгоритму, тем больше времени и памяти может потребоваться для выполнения работы. Невозможно *заранее* четко определить эту взаимосвязь, и часто приходится изменять формулировку задачи, если оказывается, что ее решение трудно описать или выполнить. А в некоторых счастливых случаях может оказаться, что найденный алгоритм может выдать информацию, более ценную, чем требовалось в первоначальной формулировке. К примеру, наша формулировка задачи связности требует лишь, чтобы мы могли определить, связана ли любая конкретная пара `p q`, но не требует демонстрации множества соединений, обеспечивающих эту связь. А такое требование усложняет задачу и приводит к другому семейству алгоритмов, которые будут рассмотрены в разделе 4.1.

Для более четкой формулировки задачи мы разработаем API-интерфейс, который инкапсулирует необходимые базовые операции: инициализация, добавление связи между двумя узлами, определение компонента, содержащего узел, определение, принадлежат ли два узла одному и тому же компоненту, и счетчик количества компонентов. Этот API-интерфейс выглядит так, как показано на рис. 1.5.3.

public class **UF**	
UF(int N)	*инициализация* N *узлов с целочисленными именами (от* 0 *до* N-1*)*
void union(int p, int q)	*добавление соединения между* p *и* q
int find(int p)	*идентификатор компонента для* p *(от* 0 *до* N-1*)*
boolean connected(int p, int q)	*возвращает* true, *если* p *и* q *принадлежат одному компоненту*
int count()	*количество компонентов*

Рис. 1.5.3. API-интерфейс объединения-поиска

Операция union() объединяет два компонента, если два узла находятся в различных компонентах, операция find() возвращает целочисленный идентификатор компонента для заданного узла, операция connected() определяет, принадлежат ли два узла одному компоненту, а метод count() возвращает количество компонентов. Все начинается с N компонентов, и каждое выполнение операции union() объединяет два различных компонента, уменьшая количество компонентов на 1.

Как мы вскоре увидим, разработка алгоритмического решения для задачи динамической связности сводится к задаче разработки реализации этого API-интерфейса. Любая реализация должна:

- определять структуру данных, представляющую известные соединения;
- содержать эффективные реализации методов union(), find(), connected() и count(), основанные на этой структуре данных

Как всегда, природа этой структуры данных непосредственно влияет на эффективность алгоритмов, поэтому структура данных и построение алгоритмов тесно взаимосвязаны. Наш API-интерфейс уже содержит соглашение, что и сайты, и компоненты идентифицируются значениями типа int от 0 до N-1, поэтому в качестве базовой структуры данных для представления компонентов имеет смысл использовать *массив, индексированный узлами* id[]. Для идентификации компонента мы будем использовать имя одного из его узлов, поэтому можно считать, что любой компонент представлен одним из своих узлов. Мы начинаем с N отдельных компонентов — каждый узел в своем отдельном компоненте — т.е. инициализируем каждый элемент id[i] значением i для всех i от 0 до N-1. И для каждого узла i мы будем хранить информацию, необходимую для определения методом find() компонента, содержащего в id[i] значение i, с помощью различных стратегий, зависящих от алгоритма. Во всех наших реализациях будет использоваться однострочная реализация метода connected(), которая возвращает логическое значение выражения find(p) == find(q).

В общем, мы начнем с алгоритма 1.5, который приведен в листинге 1.5.1. В нем используются две переменные экземпляров: счетчик компонентов и массив id[]. А реализациям find() и union() будет посвящена остальная часть данного раздела.

Листинг 1.5.1. Алгоритм 1.5. Реализация объединения-поиска

```
public class UF
{
  private int[] id;    // доступ к идентификатору компонента (индексация узлами)
  private int count;   // количество компонентов
```

```
  public UF(int N)
  {  // Инициализация массив идентификаторов компонентов.
     count = N;
     id = new int[N];
     for (int i = 0; i < N; i++)
        id[i] = i;
  }
  public int count()
  {  return count;  }
  public boolean connected(int p, int q)
  {  return find(p) == find(q);  }
  public int  find(int p)
  public void union(int p, int q)
  // См. листинги 1.5.2 (быстрый поиск), 1.5.3 (быстрое объединение)
  // и 1.5.4 (взвешенный вариант).
  public static void main(String[] args)
  {  // Решение задачи динамической связности с данными из StdIn.
     int N = StdIn.readInt();               // Ввод количества узлов.
     UF uf = new UF(N);                     // Инициализация N компонентов.
     while (!StdIn.isEmpty())
     {
        int p = StdIn.readInt();
        int q = StdIn.readInt();            // Чтение пары связанных узлов.
        if (uf.connected(p, q)) continue;   // Игнорирование, если они уже связаны.
        uf.union(p, q);                     // Объединение компонентов
        StdOut.println(p + " " + q);        //   и вывод соединения.
     }
     StdOut.println(uf.count() + " компонентов");
  }
}
```

Наша реализация UF основана на этом коде, который использует массив целых чисел id[] — такой, что метод find() возвращает одно и то же число для любого узла в каждом связном компоненте. Метод union() должен сохранять это свойство.

```
% java UF < tinyUF.txt
4 3
3 8
6 5
9 4
2 1
5 0
7 2
6 1
2 компонентов
```

Для проверки работоспособности нашего API-интерфейса и в качестве базы для разработки мы включили в main() клиент, который использует API-интерфейс для решения задачи динамической связности. Он читает значение N, а за ним — последовательность пар целых чисел (все из промежутка от 0 до N-1) и для каждой пары вызы-

вает метод find(). Если два узла в паре уже соединены, клиент переходит к чтению следующей пары, а если нет, вызывает union() и выводит эту пару. Прежде чем перейти к реализациям, мы также подготовим тестовые данные (рис. 1.5.4): файл tinyUF.txt, содержащий 11 соединений между 10 узлами (использован в небольшом примере на рис. 1.5.1), файл mediumUF.txt, содержащий 900 соединений между 625 узлами (см. пример на рис. 1.5.2), и файл largeUF.txt — пример с 2 миллионами соединений между 1 миллионом узлов. Нужно найти способ обрабатывать входные данные наподобие largeUF.txt за приемлемое время.

Для анализа алгоритмов мы будем учитывать количество обращений каждого алгоритма к элементам массива. При этом мы неявно подразумеваем, что время выполнения алгоритмов на конкретной машине прямо пропорциональны этой величине с одинаковым коэффициентом пропорциональности. Это предположение непосредственно следует из кода, его несложно экспериментально проверить, и, как мы увидим, оно представляет собой удобный способ сравнения алгоритмов.

> **Модель стоимости объединения-поиска.** При изучении алгоритмов, реализующих API-интерфейс объединения-поиска, мы будем учитывать обращения к массиву (количество обращений к элементам массива для чтения или записи).

```
% more tinyUF.txt
10
4 3
3 8
6 5
9 4
2 1
8 9
5 0
7 2
6 1
1 0
6 7

% more mediumUF.txt
625
528 503
548 523
...
[900 соединений]

% more largeUF.txt
1000000
786321 134521
696834 98245
...
[2000000 соединений]
```

Рис. 1.5.4. Тестовые файлы для задачи динамической связности

Реализации

Мы рассмотрим три различные реализации; все они основаны на использовании массива id[], индексированного узлами, для определения, принадлежат ли два узла одному и тому же связному компоненту.

Быстрый поиск

Один из способов — использование и поддержка инварианта, что узлы p и q соединены тогда и только тогда, когда id[p] равно id[q]. То есть все узлы, принадлежащие какому-то компоненту, должны иметь одинаковые значения в id[]. Этот метод называется *быстрым поиском*, т.к. метод find(p) просто возвращает значение id[p], откуда непосредственно следует, что вызов connected(p, q) сводится к проверке id[p] == id[q] и возвращает true тогда и только тогда, когда p и q принадлежат одному компоненту. Для поддержки инварианта при вызовах union(p, q) мы сначала проверяем, принадлежат ли p и q одному компоненту. Если да, то больше ничего не надо делать. Иначе оказывается, что все элементы id[], соответствующие узлам из того же компонента, что и p, содержат одно одинаковое значение, а все

Поиск сравнивает id[5] и id[9]

p q	0	1	2	3	4	5	6	7	8	9
5 9	1	1	1	8	8	1	1	1	8	8

Объединение заменяет все 1 на 8

p q	0	1	2	3	4	5	6	7	8	9
5 9	1	1	1	8	8	1	1	1	8	8
	8	8	8	8	8	8	8	8	8	8

Рис. 1.5.5. Пример быстрого поиска

p q	id[] 0	1	2	3	4	5	6	7	8	9
4 3	0	1	2	3	4	5	6	7	8	9
	0	1	2	3	3	5	6	7	8	9
3 8	0	1	2	3	3	5	6	7	8	9
	0	1	2	8	8	5	6	7	8	9
6 5	0	1	2	8	8	5	6	7	8	9
	0	1	2	8	8	5	5	7	8	9
9 4	0	1	2	8	8	5	5	7	8	9
	0	1	2	8	8	5	5	7	8	8
2 1	0	1	2	8	8	5	5	7	8	8
	0	1	1	8	8	5	5	7	8	8
8 9	0	1	1	8	8	5	5	7	8	8
5 0	0	1	1	8	8	5	5	7	8	8
	0	1	1	8	8	0	0	7	8	8
7 2	0	1	1	8	8	0	0	7	8	8
	0	1	1	8	8	0	0	1	8	8
6 1	0	1	1	8	8	0	0	1	8	8
	1	1	1	8	8	1	1	1	8	8
1 0	1	1	1	8	8	1	1	1	8	8
6 7	1	1	1	8	8	1	1	1	8	8

id[p] и id[q] различны, поэтому union() изменяет значения, равные id[p], на id[q]

id[p] и id[q] уже совпадают, поэтому ничего менять не нужно

***Рис. 1.5.6.** Трассировка быстрого поиска*

элементы id[], соответствующие узлам из того же компонента, что и q, содержат другое одинаковое значение. Для объединения этих двух компонентов в один нужно сделать так, чтобы все элементы id[], соответствующие обоим множествам узлов, имели одно и то же значение, как показано на рис. 1.5.5. Для этого выполняется перебор всего массива, и у всех элементов, равных id[p], значение меняется на id[q]. Можно, наоборот, у всех элементов, равных id[q], поменять значение на id[p] — выбор одного из этих двух вариантов произволен. Несложный код методов find() и union(), основанный на этих соображениях, приведен в листинге 1.5.2, а полная трассировка работы нашего клиента разработки на тестовом файле tinyUF.txt — на рис. 1.5.6.

Листинг 1.5.2. Быстрый поиск

```
public int find(int p)
{  return id[p];  }

public void union(int p, int q)
{  // Объединение p and q в один компонент.
   int pID = find(p);
   int qID = find(q);

   // Если p и q уже принадлежат одному
   // компоненту, ничего не надо делать.
   if (pID == qID) return;

   // Компонент p переименовывается в q.
   for (int i = 0; i < id.length; i++)
      if (id[i] == pID) id[i] = qID;
   count--;
}
```

Анализ быстрого поиска

Конечно, операция find() выполняется быстро, ведь для этого достаточно единственного обращения к массиву id[]. Но быстрый поиск обычно неприменим для больших задач, т.к. для большинства входных пар методу union() приходится перебирать весь массив id[].

Утверждение Е. Алгоритм быстрого поиска выполняет одно обращение к массиву при каждом вызове find() и от $N+3$ до $2N+1$ обращений к массиву при каждом вызове union(), который объединяет два компонента.

Доказательство непосредственно следует из кода. Каждый вызов connected() проверяет два элемента массива id[] — по разу в каждом из двух вызовов find(). Каждый вызов union() для объединения двух компонентов выполняет два вызова find(), а затем проверяет каждый из N элементов массива и изменяет от 1 до $N-1$ из них.

Предположим, к примеру, что мы используем алгоритм быстрого поиска для задачи динамической связности и дошли до единственного компонента. Для этого нужно было

выполнить не менее $N-1$ вызова union() и, следовательно, не менее $(N+3)(N-1) \sim N^2$ обращений к массиву — и мы приходим к предположению, что решение задачи динамической связности с помощью быстрого поиска может быть процессом с *квадратичным* временем. Этот анализ можно обобщить до высказывания, что алгоритм быстрого поиска является квадратичным для типичных случаев, когда, в конце концов, остается небольшое количество компонентов. Эту гипотезу легко проверить на компьютере с помощью теста с удвоением (см. характерный пример в упражнении 1.5.23). Современные компьютеры могут выполнять сотни миллионов, а то и миллиардов операций в секунду, поэтому такая трудоемкость незаметна при небольших N, но в современных приложениях может потребоваться решать задачи с миллионами или миллиардами узлов и соединений, как в нашем тестовом файле largeUF.txt. И мы приходим к неизбежному выводу, что с помощью алгоритма быстрого поиска такие задачи решить невозможно. Придется искать лучшие алгоритмы.

Быстрое объединение

Теперь мы рассмотрим алгоритм, который, наоборот, ускоряет операцию union(). Он основан на той же структуре данных — массиве id[], индексированном узлами — но значения массива будут интерпретироваться по-другому и поэтому определять более сложные структуры. А именно, элемент массива для каждого узла содержит имя другого узла в том же компоненте (или же себя) — это называется *связью* или *ссылкой*. В реализации find() мы начинаем с указанного узла, переходим по ссылке к другому узлу и т.д., пока не дойдем до *корня* — узла со ссылкой на себя (как мы увидим, это обязательно однажды произойдет). Два узла принадлежат одному и тому же компоненту тогда и только тогда, когда этот процесс приводит их к одному корню. Чтобы метод union(p, q) сохранял этот инвариант, нужно с помощью переходов по ссылкам найти корни узлов p и q, а затем переименовать один из компонентов, привязав один из корней к другому (рис. 1.5.7) — отсюда и название *быстрое объединение*. Здесь также можно произвольно выбрать переименовываемый компонент: это может быть компонент, содержащий p, или компонент, содержащий q. В реализации, приведенной в листинге 1.5.3, переименовывается тот компонент, который содержит p. На рис. 1.5.8 приведена трассировка работы алгоритма быстрого объединения с файлом tinyUF.txt. Эта трассировку лучше анализировать совместно с рис. 1.5.7, которым мы сейчас займемся более подробно.

Листинг 1.5.3. Быстрое объединение

```
private int find(int p)
{ // Поиск имени компонента.
  while (p != id[p]) p = id[p];
  return p;
}

public void union(int p, int q)
{ // Приведение p и q к общему корню.
  int pRoot = find(p);
  int qRoot = find(q);
  if (pRoot == qRoot) return;

  id[pRoot] = qRoot;

  count--;
}
```

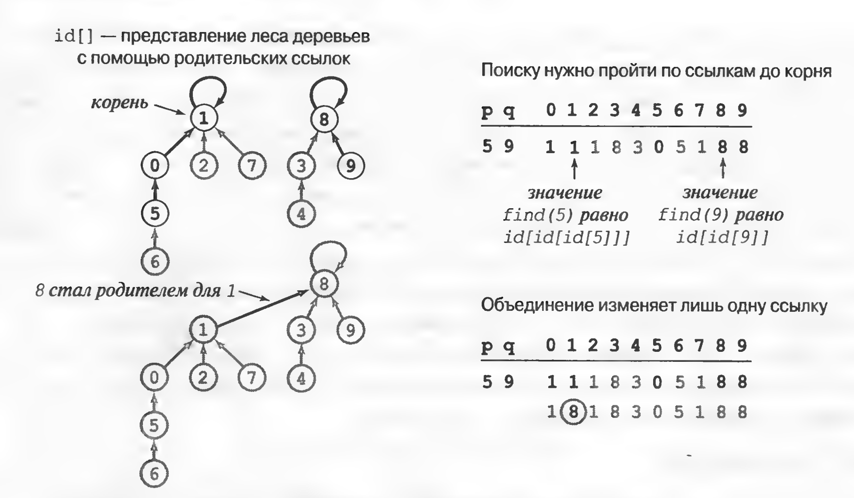


Рис. 1.5.7. Пример быстрого объединения

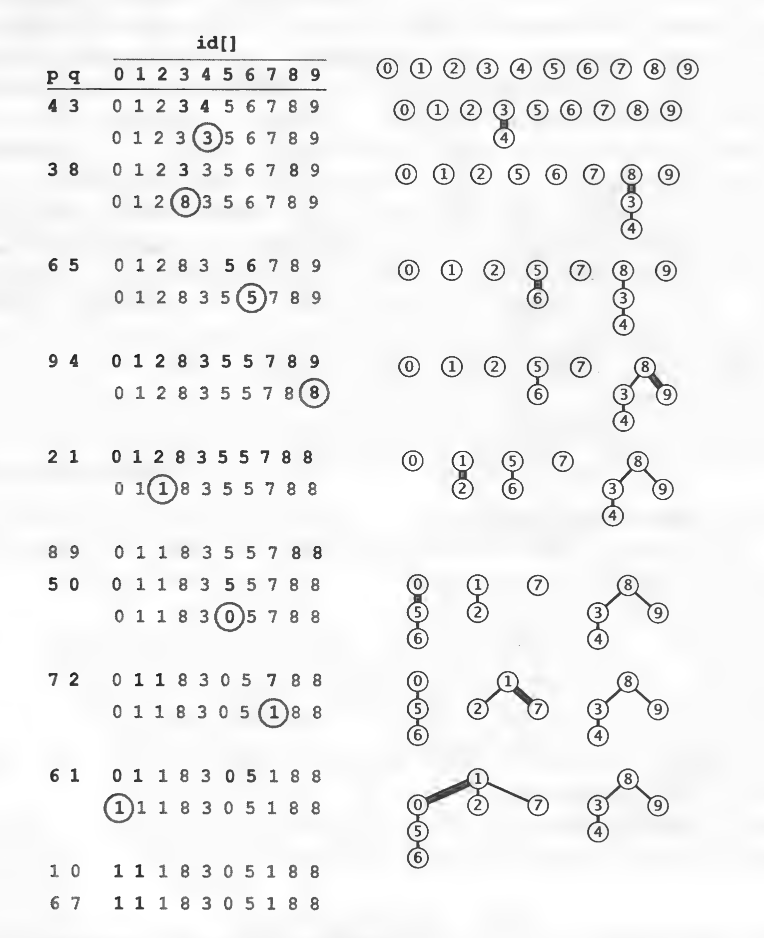


Рис. 1.5.8. Трассировка быстрого объединения (с лесами соответствующих деревьев)

Представление лесом деревьев

Код быстрого объединения краток, но не вполне нагляден. Можно графически представлять узлы в виде кружков с метками, а ссылки — в виде стрелок от одного узла к другому; такие диаграммы облегчают понимание работы алгоритма. Результирующими структурами являются *деревья*: формально массив id[] представляет лес (множество) деревьев, связанных родительскими ссылками. Для упрощения диаграмм мы обычно не будем рисовать острия стрелок (т.к. они всегда указывают вверх) и ссылки на себя в корневых узлах деревьев. Леса, соответствующие массиву id[] для файла tinyUF.txt, приведены на рис. 1.5.8.

Если начать с любого узла и пройти по ссылкам, то, в конце концов, мы дойдем до корня дерева, содержащего этот узел. Это утверждение можно доказать по индукции. Оно верно сразу после инициализации массива, т.к. каждый узел указывает на себя; а если оно верно перед выполнением операции union(), то оно, конечно, верно и после. Поэтому метод find(), представленный в листинге 1.5.3, возвращает имя корневого узла (и метод connected() может проверить, принадлежат ли два узла одному и тому же дереву). Такое представление удобно для данной задачи потому, что два узла находятся в одном дереве тогда и только тогда, когда они принадлежат одному компоненту. Кроме того, подобные деревья нетрудно создавать: реализация union() из листинга 1.5.3 объединяет два дерева одним оператором: для этого она делает корень одного дерева родителем другого.

Анализ быстрого объединения

Алгоритм быстрого объединения вроде должен работать быстрее алгоритма быстрого поиска, т.к. ему не нужно перебирать весь массив для каждой входной пары — но вот насколько быстрее? Анализ стоимости быстрого объединения сложнее, чем для быстрого поиска, поскольку эта стоимость больше зависит от природы входных данных. В лучшем случае методу find() нужно лишь одно обращение к массиву, чтобы найти идентификатор, связанный с узлом (как и в быстром поиске); а в худшем случае понадобится $2N-1$ обращений к массиву, как для узла 0 в примере на рис. 1.5.9 (это количество слишком осторожное, потому что скомпилированный код обычно *не* обращается второй раз к массиву за ссылкой id[p] в цикле while). Так что несложно подобрать входные данные для лучшего случая, когда время работы нашего клиента динамической связности будет линейным. Но также несложно подобрать входные данные и для худшего случая, который дает квадратичное время выполнения — обратите внимание на рис. 1.5.9 и утверждение Ж ниже.

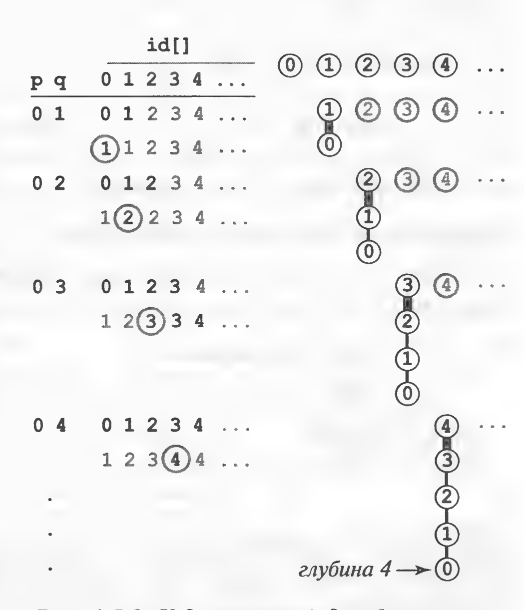


Рис. 1.5.9. *Худший случай для быстрого объединения*

К счастью, нам не нужно возиться с анализом быстрого объединения или со сравнением производительности быстрого поиска и быстрого объединения, т.к. ниже мы рас-

смотрим еще один вариант, гораздо более эффективный. Пока можно считать алгоритм быстрого объединения усовершенствованием быстрого поиска, поскольку он устраняет основной дефект быстрого поиска — операция `union()` всегда выполняется за линейное время. Эта разница, несомненно, является улучшением для типичных данных, однако быстрому объединению также присущ недостаток: мы не можем *гарантировать*, что он будет работать значительно быстрее быстрого поиска во всех случаях (для некоторых входных данных скорость быстрого объединения не превышает скорость быстрого поиска).

Определение. *Размер* дерева — это количество его узлов. *Глубина* узла в дереве — это количество ссылок на пути от него до корня. *Высота* дерева — это максимальная глубина из всех его узлов.

Утверждение Ж. Количество обращений к массиву в методе `find()` алгоритма быстрого объединения равно 1 плюс удвоенная глубина узла. Количество обращений к массиву в методах `union()` и `connected()` равно стоимости двух операций `find()` (плюс 1 для `union()`, если указанные узлы находятся в разных деревьях).

Доказательство непосредственно следует из кода.

Опять предположим, что мы используем алгоритм быстрого объединения для задачи динамической связности и приходим к одному компоненту. Непосредственное применение утверждения Ж показывает, что время выполнения в худшем случае квадратично. Допустим, пары входных данных идут в порядке `0-1`, `0-2`, `0-3` и т.д. После $N-1$ таких пар получатся N узлов, принадлежащих одному множеству, а дерево, образованное алгоритмом быстрого объединения, имеет высоту $N-1$. В этом дереве узел `0` связан с `1`, узел `1` — с `2`, `2` — с `3` и т.д. (см. диаграмму на рис. 1.5.9). Согласно утверждению Ж, количество обращений к массиву при выполнении операции `union()` для пары `0 i` равно $2i + 2$ (узел `0` имеет глубину i, а узел `i` находится на глубине 0). Поэтому общее количество обращений к массиву при выполнении операции `find()` для N пар равно $2\,(1 + 2 + \ldots + N) \sim N^2$.

Взвешенное быстрое объединение

К счастью, существует несложная модификация быстрого объединения, которая позволяет гарантировать, что такие плохие случаи не произойдут. Вместо произвольного присоединения второго дерева к первому в операции `union()` мы будем отслеживать *размер* каждого дерева и всегда присоединять меньшее дерево к большему, как на рис. 1.5.10. Для этого потребуется несколько больший объем кода и еще один массив для хранения счетчиков узлов (см. листинг 1.5.4), но это существенно повышает эффективность. Данный алгоритм мы будем называть *взвешенным быстрым объединением*. Лес деревьев, созданных этим алгоритмов для файла `tinyUF.txt`, показан на рис. 1.5.11. Даже на таком маленьком примере видно, что высота деревьев существенно меньше, чем в случае не взвешенной версии.



Рис. 1.5.10. Взвешенное быстрое объединение

Листинг 1.5.4. Алгоритм 1.5 (продолжение). Реализация объединения–поиска (взвешенный вариант)

```
public class WeightedQuickUnionUF
{
  private int[] id;     // родительская ссылка (индексация узлами)
  private int[] sz;     // размер компонента для корней (индексация узлами)
  private int count;    // количество компонентов

  public WeightedQuickUnionUF(int N)
  {
    count = N;
    id = new int[N];
    for (int i = 0; i < N; i++) id[i] = i;
    sz = new int[N];
    for (int i = 0; i < N; i++) sz[i] = 1;
  }
  public int count()
  {  return count;  }
  public boolean connected(int p, int q)
  {  return find(p) == find(q);  }
  private int find(int p)
  {  // Переходы по ссылкам до корня.
    while (p != id[p]) p = id[p];
    return p;
  }
  public void union(int p, int q)
  {
    int i = find(p);
    int j = find(q);
    if (i == j) return;
    // Меньший корень должен указывать на больший.
    if   (sz[i] < sz[j]) { id[i] = j; sz[j] += sz[i]; }
    else                 { id[j] = i; sz[i] += sz[j]; }
    count--;
  }
}
```

Этот код следует рассматривать в терминах представления лесом деревьев, описанного в тексте. Здесь добавлена еще одна переменная экземпляров — индексируемый узлами массив `sz[]`, благодаря которому метод `union()` может привязывать корень меньшего дерева к корню большего дерева. Это добавление позволяет решать задачи большого размера.

```
% java WeightedQuickUnionUF < mediumUF.txt
528 503
548 523
...
3 компонентов
% java WeightedQuickUnionUF < largeUF.txt
786321 134521
696834 98245
...
6 компонентов
```

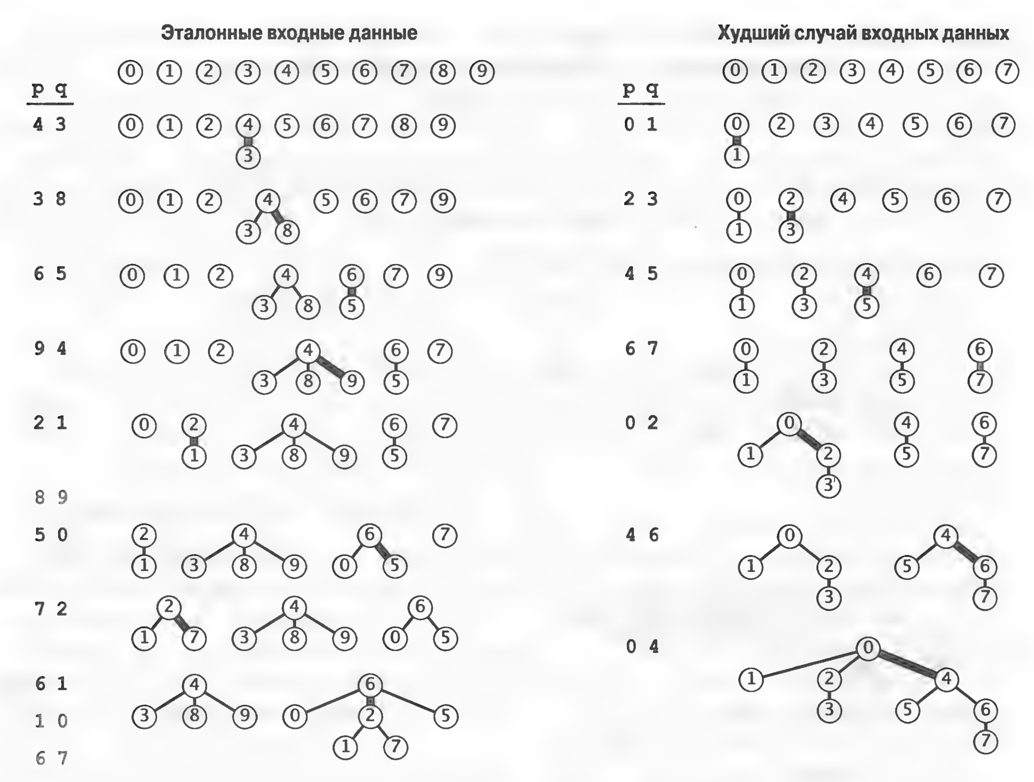


Рис. 1.5.11. Трассировки взвешенного быстрого объединения (леса деревьев)

Анализ взвешенного быстрого объединения

В правой части рис. 1.5.11 показано поведение взвешенного быстрого объединения для худшего случая — когда размеры деревьев, сливаемых методом `union()`, всегда равны (степени 2). Структуры деревьев имеют сложный вид, но у них есть простое свойство: высота дерева из 2^n узлов равна n. А при объединении двух деревьев из 2^n узлов получается дерево из 2^{n+1} узлов, т.е. высота дерева увеличивается до $n+1$. Это наблюдение позволяет доказать, что взвешенный алгоритм гарантирует *логарифмическую* производительность.

Утверждение З. Глубина любого узла в лесе, созданном взвешенным быстрым объединением для N узлов, не превышает $\lg N$.

Доказательство. Мы докажем более строгий факт с помощью (сильной) индукции: высота любого дерева размера k в лесе не превышает $\lg k$. Базовый случай следует из того, что дерево имеет высоту 0 при k, равном 1. По гипотезе индукции предположим, что высота дерева размером i не превышает $\lg i$ для всех $i < k$. При объединении дерева размером i с деревом размером j ($i \leq j$ и $i + j = k$) глубина каждого узла в меньшем множестве увеличивается на 1, но теперь они находятся в дереве размером $i + j = k$ — поэтому свойство остается верным, т.к. $1 + \lg i = \lg(i + i) \leq \lg(i + j) = \lg k$.

Следствие. Для взвешенного быстрого объединения с N узлами порядок роста для стоимости операций `find()`, `connected()` и `union()` в худшем случае равен $\log N$.

Доказательство. Каждая из этих операций выполняет не более постоянного количества обращений к массиву для каждого узла на пути от этого узла до корня в лесе.

Практическое значение утверждения 3 и следствия из него для динамической связности состоит в том, что взвешенное быстрое объединение — единственный из трех алгоритмов, который можно использовать для решения очень больших практических задач. Алгоритм взвешенного быстрого объединения выполняет *не более* $cM\lg N$ обращений к массиву для обработки M соединений между N узлами с небольшой константой c. Этот результат разительно отличается от нашего открытия, что быстрый поиск всегда (а быстрое объединение иногда) выполняет *не менее* MN обращений к массиву. Поэтому, используя взвешенное быстрое объединение, можно гарантировать возможность решения очень больших практических задач динамической связности за приемлемое время. Ценой добавления лишь нескольких строк кода мы получили программу, которая может работать в миллионы раз быстрее простых алгоритмов решения задач динамической связности, вполне возможных в реальных ситуациях.

Пример для случая 100 узлов приведен на рис. 1.5.12. Из этой диаграммы видно, что при работе взвешенного быстрого объединения на существенном расстоянии от корня находятся весьма немногие узлы. При этом часто встречаются ситуации, когда большее дерево сливается с деревом из одного узла, и тогда этот узел находится на расстоянии одной ссылки от узла. Эмпирические исследования больших задач показывают, что обычно при решении практических задач взвешенное быстрое объединение тратит *постоянное* время на выполнение одной операции. Вряд ли можно ожидать, что найдется более эффективный алгоритм.

Оптимальные алгоритмы

Можно ли найти алгоритм с *гарантированным* постоянным временем выполнения на одну операцию (табл. 1.5.1)? Это необычайно трудный вопрос занимал исследователей на протяжении многих лет.

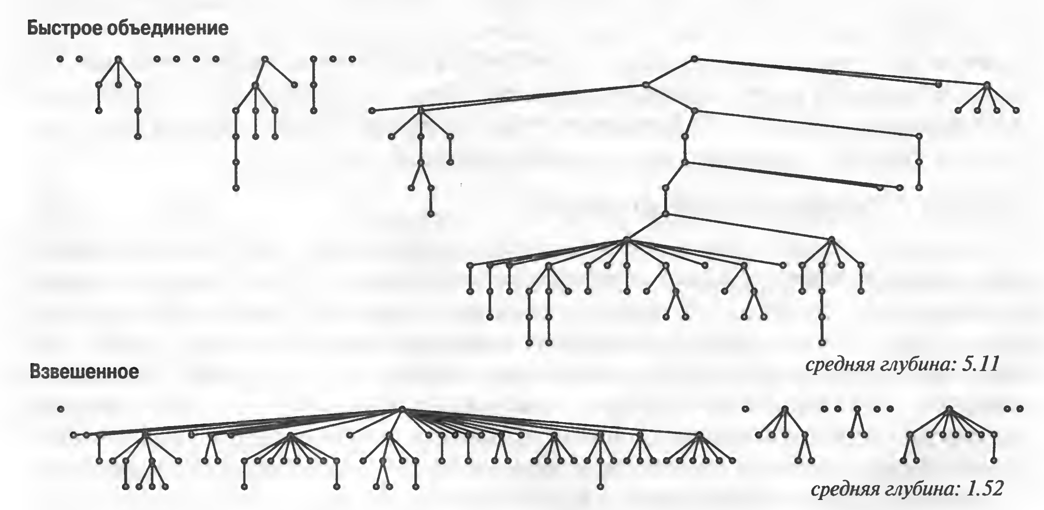


Рис. 1.5.12. *Сравнение быстрого объединения и взвешенного быстрого объединения (100 узлов, 88 операций* `union()`*)*

Таблица 1.5.1. Характеристики производительности алгоритмов объединения-поиска

Алгоритм	Порядок роста для N узлов (в худшем случае)		
	Конструктор	Объединение	Поиск
Быстрый поиск	N	N	1
Быстрое объединение	N	*высота дерева*	*высота дерева*
Взвешенное быстрое объединение	N	$\lg N$	$\lg N$
Взвешенное быстрое объединение со сжатием пути	N	*очень близко к 1, но не точно (амортизировано)* (см. упражнение 1.5.13)	
Невозможно	N	1	1

В поисках ответа был изучен ряд разновидностей быстрого объединения и его взвешенной модификации. Например, легко реализовать следующий метод *сжатия пути*. В идеале было бы замечательно, чтобы каждый узел был связан непосредственно с корнем его дерева, но не хочется платить дополнительную цену за изменение значительного количества ссылок, как в алгоритме быстрого поиска. Приблизиться к идеалу можно, привязывая к корню все узлы, которые мы *просматриваем*. Этот шаг может показаться сложным, но на самом деле реализовать его нетрудно, а структуру деревьев можно менять произвольно, если это повышает эффективность алгоритма. Для реализации сжатия пути мы добавим в метод `find()` еще один цикл, который заносит в элементы `id[]`, соответствующие всем узлам на пути к корню, ссылку непосредственно на корень. В результате деревья становятся почти полностью плоскими, т.е. приближаются к идеалу, который достигается в алгоритме быстрого поиска. Этот метод прост и эффективен, но в практических ситуациях он вряд ли даст заметное преимущество по сравнению со взвешенным быстрым объединением (см. упражнение 1.5.24). Теоретические результаты относительно данной ситуации очень сложны, но весьма примечательны. *Взвешенное быстрое объединение со сжатием пути оптимально, но требует для каждой операции не постоянного времени*. То есть не только быстрое объединение со сжатием пути не является операцией с постоянным временем в худшем случае (амортизированная трудоемкость), но и *не существует* алгоритма, который может гарантировать выполнение каждой операции объединения-поиска за амортизировано постоянное время (в условиях очень общей модели "поэлементных обращений"). Взвешенное быстрое объединение со сжатием пути весьма близко к лучшему результату для рассматриваемой задачи.

Графики амортизированной стоимости

Как и в случае реализации любого другого типа данных, лучше экспериментально проверить достоверность нашей гипотезы производительности для типичных клиентов (см. раздел 1.4). На рис. 1.5.13 показаны данные о производительности алгоритмов для нашего клиента динамической связности при решении примера определения связности для 625 узлов (`mediumUF.txt`). Генерация таких диаграмм не представляет труда (см. упражнение 1.5.16): после обработки `i`-го соединения в переменной `cost` подсчитывается количество обращений к массиву (к `id[]` или `sz[]`), а в переменной `total` — общее количество выполненных обращений к массивам. Затем выводятся серая точка с координатами `(i, cost)` и черная точка с координатами `(i, total/i)`.

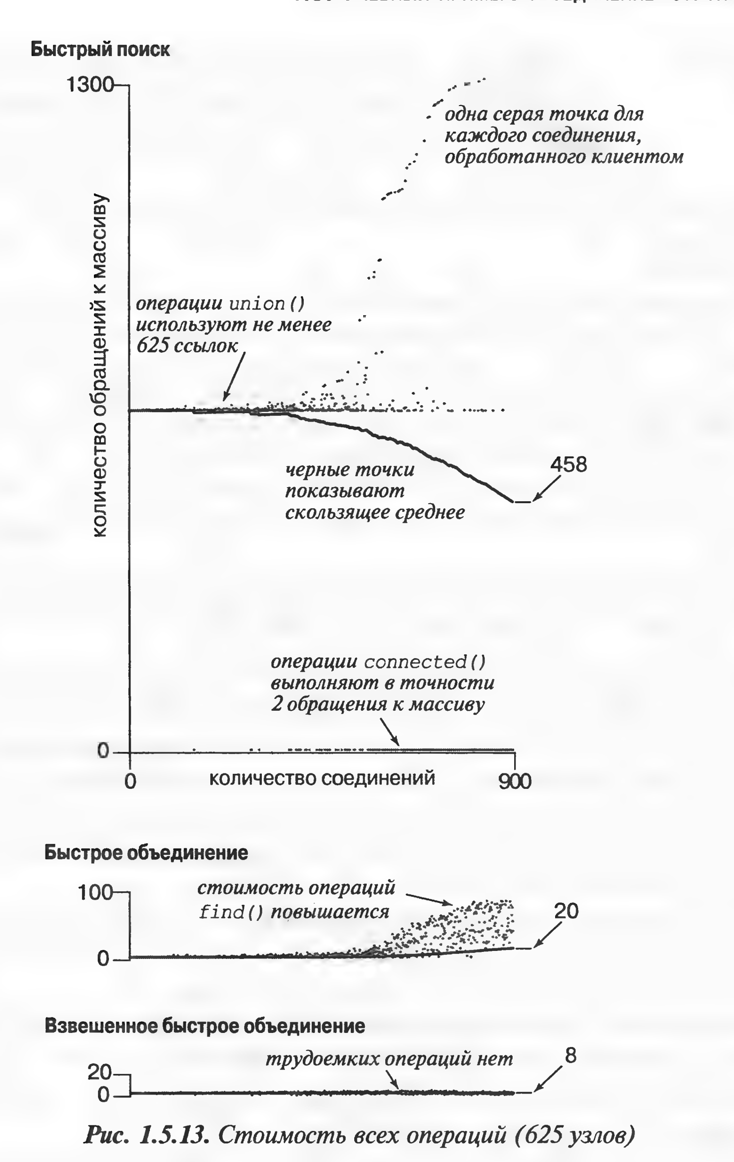


Рис. 1.5.13. *Стоимость всех операций (625 узлов)*

Черные точки отображают среднюю, т.е. амортизированную стоимость одной операции. Такие графики дают хорошее представление о поведении алгоритмов. Для *быстрого поиска* каждая операция `union()` выполняет не менее 625 обращений к массиву (плюс 1 для каждого слияния компонентов, вплоть до еще 625), а каждая операция `connected()` производит два обращения. Вначале большинство соединений приводят к вызовам `union()`, поэтому скользящее среднее находится в районе 625, но потом возникает перевес вызовов `connected()`, которые приводят к пропускам `union()`, и скользящее среднее уменьшается, хотя и остается относительно высоким. (Входные данные, которые приводят к большому количеству вызовов `connected()` и, соответственно, пропускам `union()`, дают гораздо большую производительность — см., к примеру, упражнение 1.5.23). В методе *быстрого объединения* все операции требуют вначале лишь

нескольких обращений к массиву, но потом высота деревьев становится значительной, и амортизированная стоимость заметно возрастает. В методе *взвешенного быстрого объединения* высота деревьев остается небольшой, трудоемких операций нет, а амортизированная стоимость мала. Эти эксперименты подтверждают наш вывод, что взвешенное быстрое объединение, несомненно, достойно реализации, и практически невозможно существенно превысить его производительность для практических задач.

Перспектива

Каждая из рассмотренных нами реализаций класса `UF` является усовершенствованием предыдущей в некотором интуитивном смысле, но этот процесс прошел без рывков, т.к. мы основывались на многолетнем опыте исследований в данной области. Реализации не сложны, задача четко сформулирована, что позволяет разрабатывать различные алгоритмы непосредственно при выполнении экспериментальных исследований. А сами исследования можно использовать для проверки математических формул, описывающих производительность этих алгоритмов. По возможности в этой книге мы будем выполнять такие же базовые шаги, как и для алгоритма объединения-поиска, рассмотренного в этом разделе.

- Полная и конкретная формулировка задачи, в том числе описание фундаментальных абстрактных операций, которые присущи задаче и API-интерфейсу.
- Тщательная разработка краткой реализации для примитивного алгоритма с помощью хорошо продуманного клиента разработки и реалистичных входных данных.
- Определение масштабов задач, с которыми реализация не в состоянии справиться, т.е. границ, где необходимо улучшить решение или отказаться от него.
- Разработка усовершенствованных реализаций с помощью процесса пошагового улучшения и проверка эффективности использованных идей с помощью эмпирического анализа и/или математического анализа.
- Поиск высокоуровневых абстрактных представлений структур данных или алгоритмов, которые позволяют эффективно работать с высокоуровневыми усовершенствованными версиями.
- По возможности следует гарантировать производительность в худшем случае, но обычно приемлема и хорошая производительность для типичных данных.
- Определение, когда следует оставить дальнейшие усовершенствования опытным исследователям, и переход к следующей задаче.

Возможность впечатляющего повышения производительности для подобных практических задач формирует привлекательную область изучения. Какая еще область деятельности может (в принципе) дать выигрыш в миллионы и миллиарды раз?

Разработка эффективного алгоритма — увлекательная интеллектуальная деятельность, которая может иметь непосредственную практическую отдачу. Как видно на примере задачи динамической связности, задача с простой формулировкой может привести к разработке ряда алгоритмов, которые полезны и интересны, но могут оказаться сложными и трудными для понимания. Мы ознакомимся со многими хитроумными алгоритмами, которые были созданы за многие годы для решения многих практических задач. По мере расширения сферы применимости вычислительных решений для научных и коммерческих задач растет как важность применения эффективных алгоритмов для известных задач, так и разработка эффективных алгоритмов для новых задач.

Вопросы и ответы

Вопрос. **Мне надо добавить в API-интерфейс метод `delete()`, который позволяет клиентам удалять соединения. Есть ли какие-то соображения по поводу того, как это можно сделать?**

Ответ. Никто не смог придумать алгоритм для обработки удалений, такой же простой и эффективный, как приведенные в данном разделе. Эта тема еще неоднократно будет всплывать в данной книге. Несколько структур данных, которые мы будем рассматривать, обладают свойством, что удаление каких-то элементов выполняется гораздо сложнее, чем их добавление.

Вопрос. **Что это за модель "поэлементных обращений"?**

Ответ. Модель вычислений, где в качестве затрат учитываются только обращения к памяти с произвольным доступом, достаточно большой, чтобы вместить все данные. Считается, что остальные операции выполняются без затрат.

Упражнения

1.5.1. Приведите содержимое массива `id[]` и количество обращений к массиву для каждой входной пары при обработке быстрым поиском последовательности

`9-0 3-4 5-8 7-2 2-1 5-7 0-3 4-2.`

1.5.2. Выполните упражнение 1.5.1, но для быстрого объединения (листинг 1.5.3). Кроме того, нарисуйте лес деревьев, представляемых массивом `id[]` после обработки каждой пары.

1.5.3. Выполните упражнение 1.5.1, но для взвешенного быстрого объединения (листинг 1.5.4).

1.5.4. Приведите содержимое массивов `sz[]` и `id[]` и количество обращений к массиву для каждой входной пары, соответствующей примерам в тексте для взвешенного быстрого объединения (как для эталонных входных данных, так и для худшего случая).

1.5.5. Оцените минимальное время (в днях), которое понадобится алгоритму быстрого поиска для решения задачи динамической связности с 109 узлами и 106 входными парами на компьютере, который может выполнять 109 операций в секунду. Предположите, что каждая итерация внутреннего цикла `for` требует выполнения 10 машинных инструкций.

1.5.6. Выполните упражнение 1.5.5 для взвешенного быстрого объединения.

1.5.7. Разработайте классы `QuickUnionUF` и `QuickFindUF`, которые реализуют быстрое объединение и быстрый поиск соответственно.

1.5.8. Приведите контрпример, который демонстрирует ошибочность следующей интуитивной реализации метода `union()` для быстрого поиска:

```
public void union(int p, int q)
{
   if (connected(p, q)) return;
   // Замена имени компонента p на q.
   for (int i = 0; i < id.length; i++)
      if (id[i] == id[p]) id[i] = id[q];
   count--;
}
```

1.5.9. Нарисуйте дерево, соответствующее массиву id[] на рис. 1.5.14. Может ли он получиться в результате работы взвешенного быстрого объединения? Объясните, почему это невозможно, или приведите последовательность операций, которая приводит к получению такого массива.

i	0	1	2	3	4	5	6	7	8	9
id[i]	1	1	3	1	5	6	1	3	4	5

Рис. 1.5.14. *Массив к задаче 1.5.9*

1.5.10. Предположим, что в алгоритме взвешенного быстрого объединения элементу id[find(p)] присваивается значение q, а не id[find(q)]. Будет ли полученный алгоритм работать правильно?

Ответ: да, но это увеличит высоту дерева, и не будет верна гарантия производительности.

1.5.11. Реализуйте ***взвешенный быстрый поиск***, где в элементы id[] меньшего компонента заносится идентификатор большего компонента. Как это изменение повлияет на производительность?

Творческие задачи

1.5.12. *Быстрое объединение со сжатием пути.* Добавьте в быстрое объединение (листинг 1.5.3) *сжатие пути*. Для этого добавьте в метод find() цикл, который связывает каждый сайт в пути от p до корня напрямую с корнем. Приведите последовательность входных пар, при которой этот метод строит путь длиной 4. *Примечание*: известно, что амортизированная стоимость одной операции этого алгоритма является логарифмической.

1.5.13. *Взвешенное быстрое объединение со сжатием пути.* Добавьте во взвешенное быстрое объединение (листинг 1.5.4) сжатие пути, описанное в упражнении 1.5.12. Приведите последовательность входных пар, при которой этот метод строит путь длиной 4. *Примечание*: известно, что амортизированная стоимость одной операции этого алгоритма ограничена функцией, которая называется *обратной функцией Аккермана* и которая меньше 5 для любых практически возможных значений N.

1.5.14. *Быстрое объединение, взвешенное по высоте.* Разработайте реализацию UF на основе той же стратегии, что и взвешенное быстрое объединение, но она должна отслеживать высоту дерева и всегда привязывать более короткое дерево к более высокому. Докажите, что такой алгоритм имеет логарифмическую верхнюю границу высоты деревьев.

1.5.15. *Биномиальные деревья.* Покажите, что количества узлов на каждом уровне в деревьях для худшего случая взвешенного быстрого объединения представляют собой *биномиальные коэффициенты*. Вычислите среднюю глубину узла в дереве худшего случая с $N = 2n$ узлами.

1.5.16. *Графики амортизированной стоимости.* Добавьте в реализации из упражнения 1.5.7 построение графиков амортизированной стоимости наподобие приведенных в тексте.

1.5.17. *Случайные соединения.* Разработайте клиент UF с именем ErdosRenyi, который принимает из командной строки целое значение N, генерирует случайные пары целых чисел от 0 до N-1, вызывает метод connected(), чтобы определить, соединена ли данная пара узлов, и если нет, вызывает метод union() (как в нашем клиенте разработки), а затем выводит количество сгенерированных соединений. Оформите программу в виде статического метода count(), который принимает число N в качестве аргумента, и метод main(), который принимает значение N из командной строки, вызывает count() и выводит полученный результат.

1.5.18. *Генератор случайной решетки.* Напишите программу RandomGrid, которая принимает из командной строки целое значение N, генерирует все соединения на решетке N×N, перемешивает и ориентирует их случайным образом (так что вероятности появления p q и q p равны) и выводит результат в стандартный вывод. Для случайного упорядочивания соединений используйте класс RandomBag (см. упражнение 1.3.34). Для инкапсуляции значений p и q в виде единого объекта используйте вложенный класс Connection, приведенный в листинге 1.5.5. Оформите программу в виде двух статических методов: generate(), который принимает в качестве аргумента значение N и возвращает массив соединений, и main(), который принимает значение N из командной строки, вызывает generate() и перебирает полученный массив для вывода соединений.

Листинг 1.5.5. Запись для инкапсуляции соединений

```
private class Connection
{
   int p;
   int q;
   public Connection(int p, int q)
   {  this.p = p; this.q = q;  }
}
```

1.5.19. *Анимация.* Напишите клиент RandomGrid (см. упражнение 1.5.18), который использует класс UF для проверки связности, как в нашем клиенте разработки, и применяет StdDraw для вычерчивания соединений по мере их обработки.

1.5.20. *Динамический рост.* Разработайте на основе связных списков или массивов с переменным размером реализацию взвешенного быстрого объединения, которая устраняет необходимость знать количество объектов заранее. Добавьте в API-интерфейс метод newSite(), возвращающий целочисленный идентификатор.

Эксперименты

1.5.21. *Модель Эрдеша-Реньи.* Используйте клиент из упражнения 1.5.17 для проверки гипотезы, что количество пар, которое необходимо сгенерировать для получения одного компонента, равно $\sim\frac{1}{2} N \ln N$.

1.5.22. *Тест с удвоением для модели Эрдеша-Реньи.* Разработайте клиент для тестирования производительности, который принимает из командной строки целое значение T и выполняет T повторений следующего эксперимента. Клиент из упражнения 1.5.17 генерирует случайные соединения, использует класс UF для определения связности, как в нашем клиенте разработки, продолжая работу до связывания всех узлов. Для каждого N нужно выводить значение N, среднее количество

обработанных соединений и отношение замеренного времени работы к предыдущему. Используйте эту программу для проверки высказанной в тексте гипотезы, что времена выполнения для быстрого поиска и быстрого объединения квадратичны, а время для взвешенного быстрого объединения почти линейно.

1.5.23. *Сравнение быстрого поиска с быстрым объединением для модели Эрдеша-Реньи.* Разработайте клиент для проверки производительности, который принимает из командной строки целое значение T и выполняет T повторений следующего эксперимента. Клиент из упражнения 1.5.17 генерирует случайные соединения, которые сохраняются, а затем используются для определения связности с помощью и быстрого объединения, и быстрого поиска, как в нашем клиенте разработки, продолжая работу до связывания всех узлов. Для каждого N нужно выводить значение N и отношение замеренного времени работы к предыдущему.

1.5.24. *Быстрые алгоритмы для модели Эрдеша-Реньи.* Добавьте в тесты из упражнения 1.5.23 взвешенное быстрое объединение и взвешенное быстрое объединение со сжатием пути. Сможете ли вы обнаружить различие между этими алгоритмами?

1.5.25. *Тест с удвоением для случайных решеток.* Разработайте клиент тестирования производительности, который принимает из командной строки целое значение T и выполняет T повторений следующего эксперимента. Клиент из упражнения 1.5.18 генерирует соединения на квадратной решетке N×N — в случайном порядке и со случайной ориентацией — а затем использует класс UF для определения связности, как в нашем клиенте разработки, продолжая работу до связывания всех узлов. Для каждого N нужно выводить значение N, среднее количество обработанных соединений и отношение замеренного времени работы к предыдущему. Используйте эту программу для проверки высказанной в тексте гипотезы, что времена выполнения для быстрого поиска и быстрого объединения квадратичны, а взвешенного быстрого объединения — почти линейно. *Примечание*: при удвоении N количество узлов в решетке увеличивается в 4 раза, так что следует ожидать отношение 16 для квадратичной зависимости и 4 для линейной.

1.5.26. *Амортизированная диаграмма для модели Эрдеша-Реньи.* Разработайте клиент, который принимает из командной строки целое значение N и генерирует график зависимости амортизированной стоимости всех операций в стиле графиков, приведенных в тексте. Для этого клиент должен генерировать случайные пары целых чисел от 0 до N-1, вызывать метод `connected()` для определения их связности, и если они не связаны, вызывать метод `union()` (как в нашем клиенте разработки), продолжая работу до связывания всех узлов.

Глава 2

Сортировка

Сортировка — это процесс переупорядочения последовательности объектов с целью разместить их в некотором логическом порядке. Например, отчет по движению средств на счету кредитной карточки содержит список транзакций, упорядоченный по времени, и эта упорядоченность, скорее всего, внесена каким-то алгоритмом сортировки. На заре компьютерных вычислений было распространено мнение, что до 30% всех вычислительных действий уходило на сортировку. Если в наши дни эта доля и снизилась, то в основном из-за повышения эффективности алгоритмов сортировки, а не из-за уменьшения ее важности. Повсеместное применение компьютеров может буквально завалить нас данными, и первым шагом по их организации является сортировка. Все компьютерные системы содержат реализации алгоритмов сортировки, которые используются как пользователями, так и самими системами.

Возможно, вы будете в своей деятельности пользоваться только системной сортировкой, но изучение алгоритмов сортировки важно по трем практическим причинам.

- Анализ алгоритмов сортировки представляет собой хорошее введение в сравнение производительности алгоритмов, которое мы будем использовать во всей книге.
- Похожие приемы полезны и для решения других задач.
- На основе алгоритмов сортировки разрабатываются алгоритмы для решения других задач.

Кроме того, эти алгоритмы элегантны, хрестоматийны и эффективны.

Сортировка играет самую важную роль в обработке коммерческих данных и в современных научных вычислениях. Она повсеместно применяется в обработке транзакций, комбинаторной оптимизации, астрофизике, молекулярной динамике, лингвистике, геномике, прогнозировании погоды и многих других областях. Один из алгоритмов сортировки (быстрая сортировка, см. раздел 2.3) вошел в десятку алгоритмов, наиболее важных для науки и техники в XX веке.

В данной главе мы рассмотрим несколько классических методов сортировки и эффективную реализацию фундаментального типа данных — очереди с приоритетами. Мы познакомимся с теоретическими основами для сравнения алгоритмов сортировки и завершим главу обзором применений сортировки и очередей с приоритетами.

2.1. Элементарные алгоритмы сортировки

В качестве первого знакомства с алгоритмами сортировки мы рассмотрим два элементарных метода сортировки с вариацией одного из них. Для детального изучения этих относительно простых алгоритмов есть свои причины. Во-первых, они обеспечивают контекст для изучения терминологии и базовых механизмов. Во-вторых, эти простые алгоритмы в некоторых случаях эффективнее более сложных алгоритмов, которые мы будем рассматривать после этого. В-третьих, как мы убедимся, они позволяют повысить эффективность более сложных алгоритмов.

Правила игры

Нас в первую очередь интересуют алгоритмы для переупорядочения *массивов элементов*, где каждый элемент содержит *ключ*. Цель алгоритмов сортировки — такое переупорядочение элементов, чтобы их ключи стали отсортированными в соответствии с каким-то четко определенным правилом упорядочения (обычно числовой или алфавитный порядок). Требуется переупорядочить массив так, чтобы ключ каждого элемента был не меньше ключа любого элемента с меньшим индексом и не больше ключа любого элемента с б*о*льшим индексом. Конкретные характеристики ключей и элементов могут существенно различаться в разных приложениях. В Java элементами являются просто объекты, а абстрактное понятие ключа оформлено во встроенном механизме — интерфейсе `Comparable` — о котором речь пойдет ниже в разделе "Типы данных".

Класс `Example`, приведенный в листинге 2.1.1, демонстрирует соглашения, которые мы будем использовать в дальнейшем: сортирующий код помещается в метод `sort()` наряду с приватными вспомогательными методами `less()` и `exch()` (возможно, и с другими) и клиентом `main()`. В листинге также содержится код, который может пригодиться для первоначальной отладки: клиент тестирования `main()` сортирует строки, полученные из стандартного ввода, и с помощью приватного метода `show()` выводит содержимое массива. Ниже в данной главе мы рассмотрим различные клиенты тестирования для сравнения алгоритмов и для изучения их производительности. Для различения методов сортировки мы будем по-разному называть наши различные классы сортировки. Поэтому клиенты могут вызывать различные реализации по их именам: `Insertion.sort()`, `Merge.sort()`, `Quick.sort()` и т.д.

Листинг 2.1.1. Шаблон для классов сортировки

```
public class Example
{
   public static void sort(Comparable[] a)
   {  /* См. алгоритм 2.1, 2.2, 2.3, 2.4, 2.5 или 2.7. */  }

   private static boolean less(Comparable v, Comparable w)
   {  return v.compareTo(w) < 0;  }

   private static void exch(Comparable[] a, int i, int j)
   {  Comparable t = a[i]; a[i] = a[j]; a[j] = t;  }
```

```
  private static void show(Comparable[] a)
  {  // Вывод массива в одной строке.
     for (int i = 0; i < a.length; i++)
        StdOut.print(a[i] + " ");
     StdOut.println();
  }
  public static boolean isSorted(Comparable[] a)
  {  // Проверка упорядоченности элементов массива.
     for (int i = 1; i < a.length; i++)
        if (less(a[i], a[i-1]))  return false;
     return true;
  }
  public static void main(String[] args)
  {  // Чтение строк из стандартного ввода, их сортировка и вывод.
     String[] a = In.readStrings();
     sort(a);
     assert isSorted(a);
     show(a);
  }
}
```

Этот класс демонстрирует наши соглашения по реализации сортировки массивов. Для каждого рассматриваемого алгоритма сортировки мы добавим метод `sort()` в класс вроде `Example`, только с именем, соответствующим алгоритму. Клиент тестирования сортирует строки, полученные из стандартного ввода, но наши методы сортировки пригодны для любого типа данных, который реализует интерфейс `Comparable`.

```
% more tiny.txt
S O R T E X A M P L E

% java Example < tiny.txt
A E E L M O P R S T X
```

```
% more words3.txt
bed bug dad yes zoo ... all bad yet

% java Example < words.txt
all bad bed bug dad ... yes yet zoo
```

За несколькими исключениями наш код сортировки обращается к данным с помощью лишь двух операций: метода `less()` для сравнения элементов и метода `exch()` для их обмена. Метод `exch()` нетрудно реализовать самостоятельно, а реализацию метода `less()` облегчает интерфейс `Comparable`. Ограничение доступа к данным только этими двумя операциями делает наш код более понятным и переносимым, а также облегчает проверку корректности, изучение производительности и сравнение алгоритмов. Но прежде чем перейти к изучению реализаций сортировки, мы обсудим ряд моментов, которые важны при рассмотрении любого метода сортировки.

Проверка

Всегда ли реализация сортировки упорядочивает элементы массива вне зависимости от первоначального порядка? В качестве меры предосторожности мы используем

в нашем клиенте тестирования оператор `assert isSorted(a);` — он проверяет, что после сортировки элементы массива действительно упорядочены. Имеет смысл включать этот оператор в *любую* реализацию сортировки, хотя обычно мы проверяем работоспособность кода и математически доказываем корректность алгоритмов. Эта проверка достаточна лишь тогда, когда для обмена элементов массива применяется только метод `exch()`. Если код непосредственно записывает значения в элементы массива, у нас нет полной уверенности (например, код, который перезаписывает на место исходного массива одно и то же значение, пройдет такую проверку).

Время выполнения

Кроме работоспособности, мы будем проверять *производительность* алгоритмов. Вначале мы определим количество базовых операций (сравнений и обменов, или, возможно, количество обращений к массиву для чтения или записи), которые выполняют различные алгоритмы сортировки для разных естественных моделей входных данных. Затем мы используем эти факты для выдвижения гипотез о сравнительной производительности алгоритмов и предоставим средства для экспериментальной проверки верности этих гипотез. Мы будем использовать однотипный стиль кодирования, чтобы облегчить разработку верных гипотез о производительности, которые будут верны в типичных случаях.

Модель стоимости сортировки. При изучении алгоритмов сортировки мы подсчитываем *сравнения* и *обмены*. Для алгоритмов, которые не используют обмены, мы будем подсчитывать *обращения к массиву*.

Дополнительная память

Объем дополнительной памяти, необходимой алгоритму сортировки, часто не менее важен, чем время выполнения. В этом смысле алгоритмы сортировки делятся на два базовых типа: те, которые выполняют сортировку *на месте* и не используют дополнительную память (кроме, возможно, небольшого функционального стека или постоянного количества переменных экземпляров), и те, которым нужна дополнительная память, достаточная для хранения копии сортируемого массива.

Типы данных

Наш код сортировки одинаково эффективен для любых типов элементов, которые реализуют интерфейс `Comparable`. Такое следование соглашению, принятому в Java, удобно тем, что многие типы данных, которые приходится упорядочивать, реализуют этот интерфейс. Например, интерфейс `Comparable` реализуют типы-оболочки для чисел, предусмотренные в Java, такие как `Integer` и `Double` — а также `String` и более сложные типы вроде `File` или `URL`. Поэтому можно просто вызвать один из наших методов сортировки с массивом любых таких типов в качестве аргумента. Например, код в листинге 2.1.2 использует быструю сортировку (см. раздел 2.3) для сортировки `N` случайных значений типа `Double`. Если нам понадобится создать собственный тип, мы можем позволить клиентскому коду упорядочивать значения этого типа данных, реализовав интерфейс `Comparable`. Для этого нужно лишь реализовать метод `compareTo()`, который определяет упорядочение объектов данного типа — *естественный порядок* для этого типа — как показано в листинге 2.1.3 для нашего типа `Date` (рис. 1.2.22).

Листинг 2.1.2. Сортировка массива случайных значений

```
Double a[] = new Double[N];
for (int i = 0; i < N; i++)
   a[i] = StdRandom.uniform();
Quick.sort(a);
```

Листинг 2.1.3. Определение сравнимого типа

```
public class Date implements Comparable<Date>
{
   private final int day;
   private final int month;
   private final int year;
   public Date(int d, int m, int y)
   {  day = d; month = m; year = y; }
   public int day()     {  return day;    }
   public int month()   {  return month;  }
   public int year()    {  return year;   }
   public int compareTo(Date that)
   {
      if (this.year  > that.year ) return +1;
      if (this.year  < that.year ) return -1;
      if (this.month > that.month) return +1;
      if (this.month < that.month) return -1;
      if (this.day   > that.day  ) return +1;
      if (this.day   < that.day  ) return -1;
      return 0;
   }
   public String toString()
   { return month + "/" + day + "/" + year; }
}
```

По соглашению, принятому в Java, вызов `v.compareTo(w)` возвращает целое число, которое меньше нуля, равно нулю или больше нуля (обычно -1, 0 или +1), если v < w, v = w или v > w соответственно. Для экономии места до конца этого раздела мы будем вместо выражения `v.compareTo(w)>0` использовать сокращение v > w. По соглашению вызов `v.compareTo(w)` должен генерировать исключение тогда и только тогда, когда тип v и w не является сравнимым или одно из значений равно `null`. Более того, метод `compareTo()` должен реализовывать *полную упорядоченность*, т.е. операцию со следующими свойствами:

- *рефлексивность* — для любого v верно, что v = v;
- *антисимметричность* — для любых v и w верно, что если v < w, то w > v, а если v = w, то w = v;
- *транзитивность* — для любых u, v и w верно, что если u <= v и v <= w, то u <= w.

Эти правила интуитивно понятны и общеприняты в математике, и поэтому их нетрудно применять. В общем, метод `compareTo()` реализует абстракцию *ключа*: он определяет упорядоченность сортируемых элементов (объектов), которые могут иметь любой тип, если он реализует интерфейс `Comparable`. Обратите внимание, что метод `compareTo()` не обязан использовать все переменные экземпляров. Ключ может занимать небольшую часть каждого элемента.

До конца настоящей главы мы будем рассматривать многочисленные алгоритмы сортировки объектов, для которых существует естественный порядок. С целью сравнения алгоритмов мы ознакомимся с рядом их свойств, в том числе и с количеством сравнений и обменов, используемых этими алгоритмами для различных видов данных, и с объемом дополнительной памяти, необходимой для их работы. Эти свойства приведут к формулировке гипотез о производительности, многие из которых проверены на бесчисленном множестве компьютеров на протяжении нескольких последних десятков лет. Всегда необходимо проверять работоспособность конкретных реализаций, поэтому мы подумаем и над необходимыми для этого инструментами. После изучения классических алгоритмов — сортировки выбором, сортировки вставками, сортировки Шелла, сортировки слиянием, быстрой сортировки и пирамидальной сортировки — в разделе 2.5 мы рассмотрим практические применения и возможные сложности.

Сортировка выбором

Один из простейших алгоритмов сортировки работает так. Сначала в массиве находится наименьший элемент, и он меняется местами с первым элементом (возможно, и с собой, если первый элемент как раз и есть наименьший). Потом находится следующий наименьший элемент и меняется местами со вторым элементом. Подобное продолжается до упорядочивания всего массива. Этот метод называется ***сортировкой выбором***, т.к. при его работе многократно выбирается наименьший из оставшихся элемент.

Как видно из реализации (см. алгоритм 2.1 в листинге 2.1.4), внутренний цикл сортировки выбором просто выполняет сравнения текущего элемента с найденным к этому моменту наименьшим элементом (плюс код, необходимый для наращивания текущего индекса и проверки, не вышел ли он за границу массива) — проще не придумаешь. Перемещение элементов выполняется за пределами внутреннего цикла: каждый обмен помещает один элемент в его окончательную позицию, и, значит, количество обменов равно N. Поэтому время выполнения в основном обусловливается количеством сравнений.

Листинг 2.1.4. Алгоритм 2.1. Сортировка выбором

```
public class Selection
{
  public static void sort(Comparable[] a)
  { // Сортировка a[] по возрастанию.
    int N = a.length;                    // длина массива
    for (int i = 0; i < N; i++)
    { // Перестановка a[i] с наименьшим элементом из a[i+1...N].
      int min = i;                       // индекс минимального элемента.
      for (int j = i+1; j < N; j++)
        if (less(a[j], a[min])) min = j;
      exch(a, i, min);
    }
  }
  // Реализации less(), exch(), isSorted() и main() см. в листинге 2.1.1.
}
```

Для каждого `i` данная реализация помещает `i`-й наименьший элемент в `a[i]`. Элементы слева от позиции `i` — это `i` наименьших элементов массива, и их уже просматривать не надо.

Утверждение А. Для упорядочения массива длиной N сортировка выбором использует $\sim N^2/2$ сравнений и N перестановок.

Доказательство. Этот факт следует из внешнего вида трассировки (рис. 2.1.1) — таблицы $N \times N$, в которой яркие буквы соответствуют сравнениям. Примерно половина элементов таблицы обозначена более темными буквами — это те, которые находятся на диагонали и над ней. Элементы на диагонали соответствуют обменам значениями. Если точнее просмотреть код, то для каждого i от 0 до $N-1$ выполняется одна перестановка и $N-1-i$ сравнений, поэтому всего получается N перестановок и $(N-1) + (N-2) + \ldots + 2 + 1 + 0 = N(N-1)/2 \sim N^2/2$ сравнений.

		a[]										
i	min	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
0	6	S	O	R	T	E	X	**A**	M	P	L	E
1	4	A	O	R	T	**E**	X	S	M	P	L	E
2	10	A	E	R	T	O	X	S	M	P	L	**E**
3	9	A	E	E	T	O	X	S	M	P	**L**	R
4	7	A	E	E	L	O	X	S	**M**	P	T	R
5	7	A	E	E	L	M	X	S	**O**	P	T	R
6	8	A	E	E	L	M	O	S	X	**P**	T	R
7	10	A	E	E	L	M	O	P	X	S	T	**R**
8	8	A	E	E	L	M	O	P	R	**S**	T	X
9	9	A	E	E	L	M	O	P	R	S	**T**	X
10	10	A	E	E	L	M	O	P	R	S	T	**X**
		A	E	E	L	M	O	P	R	S	T	X

эти элементы просматриваются при поиске минимума

эти элементы — a[min]

эти элементы находятся в окончательной позиции

Рис. 2.1.1. *Трассировка сортировки выбором (содержимое массива сразу после каждой перестановки)*

Итак, сортировка выбором представляет собой простой метод сортировки, который несложно понять и реализовать, и который обладает следующими двумя характерными свойствами.

Время выполнения не чувствительно к входным данным

Процесс поиска наименьшего элемента на одном проходе по массиву не дает существенной информации о том, где может находиться наименьший элемент на следующем проходе. В некоторых случаях это свойство может оказаться очень неудобным. Например, пользователь, использующий клиент сортировки, может сильно удивиться, что сортировка выбором работает с уже упорядоченным массивом или с массивом со всеми равными ключами так же долго, как и со случайно упорядоченным массивом. Как мы увидим, другие алгоритмы лучше используют начальную упорядоченность входных данных.

Перемещение данных минимально

Каждая их N операций обмена изменяет значения двух элементов массива, поэтому сортировка выбором использует N обменов — количество обращений к массиву является *линейной* функцией от размера массива. Ни у одного из других алгоритмов сортировки, которые мы будем рассматривать, нет такого свойства (обычно линейно-логарифмическая или квадратичная зависимость).

Сортировка вставками

Алгоритм, который часто применяют игроки в бридж для упорядочения своих карт, состоит в следующем: они просматривают карты поочередно и вставляют каждую из них на свое место среди уже просмотренных (сохраняя упорядоченность). В компьютерной реализации потребуется место для вставки текущего элемента — для этого перед вставкой необходимо сдвинуть бо́льшие элементы на одну позицию вправо, а затем вставить текущий элемент на освободившееся место. Этот метод реализован в алгоритме 2.2 (см. листинг 2.1.5), который называется *сортировкой вставками*.

Листинг 2.1.5. Алгоритм 2.2. Сортировка вставками

```
public class Insertion
{
  public static void sort(Comparable[] a)
  { // Сортировка a[] по возрастанию.
    int N = a.length;
    for (int i = 1; i < N; i++)
    { // Вставка a[i] среди a[i-1], a[i-2], a[i-3], ... .
      for (int j = i; j > 0 && less(a[j], a[j-1]); j--)
        exch(a, j, j-1);
    }
  }
  // Реализации less(), exch(), isSorted() и main() см. в листинге 2.1.1.
}
```

Для каждого `i` от `0` до `N-1` элемент `a[i]` обменивается с элементами из `a[0]...a[i-1]`, которые меньше его. По мере прохода индекса `i` слева направо элементы массива слева от него становятся упорядоченными, и при достижении правого конца упорядоченным становится весь массив.

Как и в сортировке выбором, элементы слева от текущего индекса упорядочены в процессе сортировки, но они не (обязательно) находятся в своих окончательных позициях, т.к. могут быть позже сдвинуты, чтобы освободить место для найденного меньшего элемента. Но после завершения пробега индекса массив оказывается полностью упорядоченным.

В отличие от сортировки выбором, время выполнения сортировки вставками зависит от начального упорядочения входных данных. Например, если элементы большого массива уже упорядочены (или почти упорядочены), то сортировка вставками выполняется *значительно* быстрее, чем в случае первоначального случайного или обратного упорядочения.

Утверждение Б. Для сортировки случайно упорядоченного массива длиной N с различными ключами сортировка вставками в среднем использует $\sim N^2/2$ сравнений и $\sim N^2/2$ перестановок, а в лучшем случае — $N-1$ сравнений и 0 перестановок.

Доказательство. Как и в случае утверждения А, количество сравнений и перестановок легко наглядно увидеть на диаграмме $N \times N$, демонстрирующей процесс сортировки (рис. 2.1.2). Мы подсчитываем элементы под диагональю — все элементы в худшем случае или ни один в лучшем. Для случайно упорядоченных файлов можно ожидать, что каждый элемент в среднем сдвигается назад примерно наполовину, поэтому мы считаем половину элементов под диагональю.

Количество сравнений равно количеству перестановок плюс дополнительное слагаемое, равное N минус количество раз, когда вставляемый элемент был наименьшим из рассматриваемых. В худшем случае (обратное упорядочение) это слагаемое пренебрежимо мало по сравнению с общей суммой; в лучшем случае (уже упорядоченный массив) оно равно $N-1$.

эти элементы — a[j]

эти элементы не перемещаются

эти элементы сдвигаются на одну позицию вправо для вставки

		a[]										
i	j	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
1	0	**O**	S	R	T	E	X	A	M	P	L	E
2	1	O	**R**	S	T	E	X	A	M	P	L	E
3	3	O	R	S	**T**	E	X	A	M	P	L	E
4	0	**E**	O	R	S	T	X	A	M	P	L	E
5	5	E	O	R	S	T	**X**	A	M	P	L	E
6	0	**A**	E	O	R	S	T	X	M	P	L	E
7	2	A	E	**M**	O	R	S	T	X	P	L	E
8	4	A	E	M	O	**P**	R	S	T	X	L	E
9	2	A	E	**L**	M	O	P	R	S	T	X	E
10	2	A	E	**E**	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

Рис. 2.1.2. *Трассировка сортировки вставками (содержимое массива сразу после каждой вставки)*

Сортировка вставками хорошо работает для некоторых типов неслучайных массивов (даже и больших), которые часто встречаются на практике. Например, как было замечено, подумайте, что произойдет, если использовать сортировку вставками для массива, который уже упорядочен.

Сразу же обнаруживается, что каждый элемент находится на своем месте в массиве, и общее время выполнения оказывается линейным. (Время сортировки выбором для такого массива квадратично.) Так же дело обстоит и если все ключи одинаковы — отсюда и условие в утверждении Б, что все ключи должны быть различны.

Можно рассмотреть концепцию частично упорядоченного массива. *Инверсия* — это пара элементов, которые нарушают упорядоченность в массиве. Например, в последовательности `E X A M P L E` имеется 11 инверсий: `E-A`, `X-A`, `X-M`, `X-P`, `X-L`, `X-E`, `M-L`, `M-E`, `P-L`, `P-E` и `L-E`. Если количество инверсий в массиве меньше, чем размер массива с некоторым постоянным множителем, то такой массив называется *частично упорядоченным*. Вот типичные примеры частично упорядоченных массивов:

- массив, каждый элемент которого находится недалеко от своей окончательной позиции;
- небольшой массив, добавленный к большому отсортированному массиву;
- массив, в котором лишь несколько элементов находятся не на месте.

Для таких массивов сортировка вставками вполне эффективна — в отличие от сортировки выбором. Вообще говоря, при небольшом количестве инверсий сортировка вставками может работать быстрее любого другого метода сортировки из рассматриваемых в данной главе.

Утверждение В. Количество перестановок, используемых сортировкой вставками, равно количеству инверсий в массиве, а количество сравнений не меньше количества инверсий и не больше количества инверсий плюс размер массива минус 1.

Доказательство. В каждой перестановке задействованы два инвертированных смежных элемента, при этом количество инверсий уменьшается на единицу, а в упорядоченном массиве количество инверсий снижается до нуля. Каждой перестановке соответствует сравнение, и могут быть дополнительные сравнения для каждого `i` от `1` до `N-1` (когда `a[i]` не доходит до левого края массива).

Сортировку вставками нетрудно существенно ускорить, укоротив внутренний цикл: нужно просто сдвигать бо́льшие элементы вправо на одну позицию, а не выполнять полные перестановки (при этом количество обращений к массиву уменьшается вдвое). Это усовершенствование мы оставляем читателям в качестве упражнения (упражнение 2.1.25).

Итак, сортировка вставками — замечательный метод для частично упорядоченных массивов, а также вполне приемлемый метод для небольших массивов. Эти факты важны не только потому, что такие массивы часто встречаются на практике, но и потому, что оба таких вида массивов возникают на промежуточных этапах более сложных алгоритмов сортировки. Поэтому сортировка вставками еще будет рассматриваться при изучении этих алгоритмов.

Визуализация алгоритмов сортировки

На протяжении всей этой главы мы будем использовать простое визуальное представление, которое поможет нам описывать свойства алгоритмов сортировки. Вместо наблюдения за процессом сортировки таких значений ключей, как буквы, числа или слова, мы будем использовать вертикальные полоски, которые необходимо упорядочить по их длинам. Такое представление позволяет лучше понять поведение метода в процессе сортировки.

Например, глядя на рис. 2.1.3, можно сразу увидеть, что сортировка вставками не затрагивает элементы справа от указателя перебора, а сортировка выбором не затрагивает элементы слева от указателя перебора. Более того, наглядно видно, что сортировка вставками также не затрагивает элементы, меньшие вставляемого, и поэтому количество используемых ей сравнений в среднем примерно вдвое меньше количества сравнений при сортировке выбором.

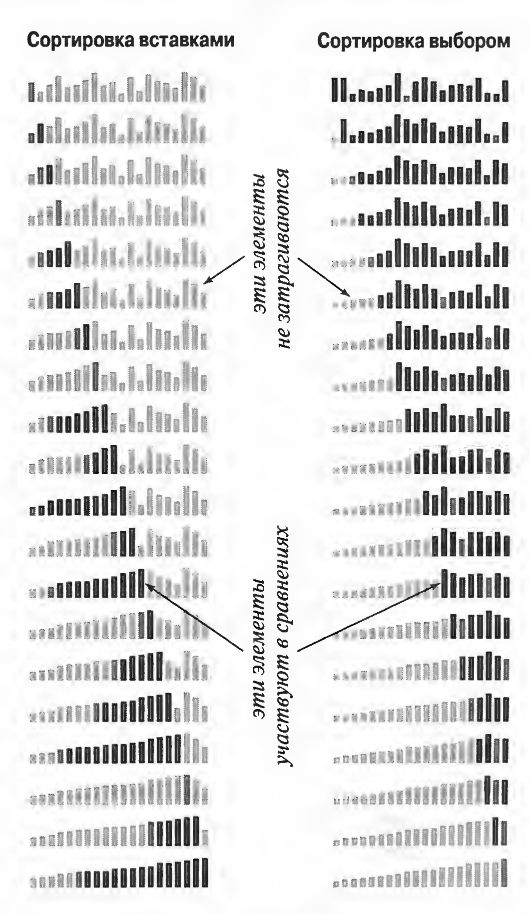


Рис. 2.1.3. Визуальные трассировки элементарных алгоритмов сортировки

Используя нашу библиотеку `StdDraw`, выполнить визуальную трассировку не намного сложнее, чем стандартную трассировку. Мы сортируем значения `Double`, добавляем в алгоритм вызов `show()` (как и для стандартной трассировки) и используем версию `show()`, которая выводит полоски с помощью `StdDraw`, а не текстовый результат. Сложнее всего задать масштаб по оси Y, чтобы строки трассировки находились в естественном порядке. Рекомендуем вам проработать упражнение 2.1.18, чтобы лучше понять ценность визуальных трассировок и легкость их создания.

Еще проще выполнить *анимированную* трассировку, чтобы увидеть процесс сортировки массива в динамике. Для создания анимированной трассировки нужен, по сути, тот же процесс, что и в предыдущем абзаце, но без забот об оси Y — достаточно каждый раз очищать окно и перерисовывать полоски. Такие трассировки невозможно поместить на печатную страницу, но они отлично помогают понять принцип работы алгоритма. Рекомендуем вам удостовериться в этом самостоятельно, выполнив упражнение 2.1.17.

Сравнение двух алгоритмов сортировки

Теперь, когда у нас есть две реализации, естественно поинтересоваться, которая из них быстрее — сортировка выбором (алгоритм 2.1) или сортировка вставками (алгоритм 2.2). Подобные вопросы постоянно возникают при изучении алгоритмов и, конечно, в настоящей книге. Некоторые фундаментальные принципы были изложены в главе 1, но мы воспользуемся этим первым случаем, чтобы продемонстрировать базовый подход к ответам на подобные вопросы. Как правило, мы следуем подходу, описанному в разделе 1.4, и сравниваем алгоритмы с помощью таких средств:

- реализация алгоритмов и их отладка;
- анализ их базовых свойств;
- формулировка гипотезы об относительной производительности;
- экспериментальная проверка гипотезы.

Эти шаги — просто проверенный временем *научный подход*, примененный к изучению алгоритмов.

В данном контексте алгоритмы 2.1 и 2.2 представляют первый шаг; утверждения А, Б и В составляют второй шаг; свойство Г (ниже) составляет третий шаг; а класс `SortCompare` (см. листинг 2.1.7) позволяет выполнить четвертый шаг. Все эти действия взаимосвязаны.

Наши краткие описания скрывают тот объем усилий, который потребовался для корректной реализации, анализа и тестирования алгоритмов. Каждый программист знает, что такой код представляет собой продукт длительного цикла отладки и усовершенствований, каждый математик знает, что четкий анализ может оказаться очень трудным, а каждый ученый знает, что формулировка гипотез и обдумывание и выполнение экспериментов для их проверки требуют большой осторожности. Полная разработка подобных результатов — занятие для экспертов, изучающих наиболее важные из рассматриваемых здесь алгоритмов, но каждому программисту, использующему такой алгоритм, следует знать о научном обосновании свойств его производительности.

После разработки реализаций следующим шагом необходимо выбрать подходящую модель входных данных. Для сортировки естественной моделью, которая уже использовалась в утверждениях А, Б и В, является предположение, что массивы случайно упоря-

дочены *и* что значения всех ключей различны. В тех приложениях, где может появиться значительное количество одинаковых ключей, потребуется более сложная модель.

Как сформулировать гипотезу о времени выполнения сортировки вставками и сортировки выбором для случайно упорядоченных массивов? Из алгоритмов 2.1 и 2.2 и утверждений А и Б непосредственно следует, что время выполнения обоих алгоритмов для случайно упорядоченных массивов должно быть квадратичным. То есть время работы сортировки вставками для таких данных пропорционально N^2 с небольшим постоянным коэффициентом пропорциональности, а время выполнения сортировки выбором пропорционально N^2 с другим небольшим коэффициентом пропорциональности. Величины этих коэффициентов зависят от стоимости сравнений и перестановок на конкретном компьютере. Для многих типов данных и для типичных компьютеров можно предположить, что эти стоимости сопоставимы (хотя мы еще столкнемся с несколькими существенными исключениями). Из наших рассуждений непосредственно следует такая гипотеза.

Свойство Г. Значения времени выполнения сортировки вставками и сортировки выбором квадратичны и сопоставимы друг с другом (с постоянным коэффициентом, не существенно отличающимся от 1) для случайно упорядоченных массивов различных значений.

Обоснование. За последние полвека это заявление проверено на множестве различных компьютеров. Сортировка вставками была примерно вдвое быстрее сортировки выбором в 1980 г., когда было написано первое издание этой книги, и это справедливо в настоящее время, хотя тогда для сортировки 100 000 элементов этим алгоритмам требовалось несколько часов, а сейчас — несколько секунд. Но работает ли сортировка вставками (немного) быстрее, чем сортировка выбором, на вашем компьютере? Чтобы узнать это, можно воспользоваться классом `SortCompare`, приведенным в листинге 2.1.6, который применяет методы `sort()` из классов, указанных в аргументах командной строки, для проведения заданного количества экспериментов (сортировка массивов заданного размера), а затем выводит отношение замеренных времен выполнения этих алгоритмов.

Листинг 2.1.6. Сравнение двух алгоритмов сортировки

```
public class SortCompare
{
  public static double time(String alg, Double[] a)
  {  /* См. листинг 2.1.7. */  }

  public static double timeRandomInput(String alg, int N, int T)
  {  alg указывает алгоритм сортировки T случайных массивов длиной N.
     double total = 0.0;
     Double[] a = new Double[N];
     for (int t = 0; t < T; t++)
     {  // Выполнение одного эксперимента (генерация и сортировка массива).
        for (int i = 0; i < N; i++)
           a[i] = StdRandom.uniform();
        total += time(alg, a);
     }
     return total;
  }
```

```
  public static void main(String[] args)
  {
    String alg1 = args[0];
    String alg2 = args[1];
    int N = Integer.parseInt(args[2]);
    int T = Integer.parseInt(args[3]);
    double t1 = timeRandomInput(alg1, N, T);  // общее время для alg1
    double t2 = timeRandomInput(alg2, N, T);  // общее время для alg2
    StdOut.printf("Для %d случайных Doubles\n   %s в", N, alg1);
    StdOut.printf(" %.1f раз быстрее, чем %s\n", t2/t1, alg2);
  }
}
```

Этот клиент выполняет две различные сортировки, указанные в первых двух аргументах командной строки, для массивов размером N (третий аргумент) случайных значений Double от 0.0 до 1.0, и повторяет такой эксперимент T раз (четвертый аргумент), а затем выводит отношение суммарных значений времени выполнения.

```
% java SortCompare Вставки Выбор 1000 100
Для 1000 случайных Doubles
   Вставки в 1.7 раз быстрее, чем Выбор
```

Для проверки этой гипотезы мы выполнили эксперименты с использованием класса SortCompare (см. листинг 2.1.6). Как обычно, время выполнения замеряется с помощью экземпляра Stopwatch. Реализация метода time(), приведенная в листинге 2.1.7, выполняет все необходимое для базовых алгоритмов сортировки, описанных в данной главе. Модель “случайно упорядоченных” входных данных встроена в метод timeRandomInput() класса SortCompare, который генерирует случайные значения Double, сортирует их и возвращает общее замеренное время сортировки для указанного количества проб. Использование случайных значений Double от 0.0 до 1.0 гораздо проще, чем применение библиотечной функции вроде StdRandom.shuffle() — да и эффективнее, поскольку появление равных ключей очень маловероятно (см. упражнение 2.5.31). Как было сказано в главе 1, количество проб, передаваемое через аргумент, предназначено для использования закона больших чисел (чем больше проб, тем точнее суммарное время выполнения, деленное на количество проб, больше приближается к истинному среднему времени выполнения) и для нивелирования системных эффектов. Мы рекомендуем поэкспериментировать с классом SortCompare на своем компьютере, чтобы проверить, до какой степени верен вывод об устойчивости сортировки вставками и сортировки выбором.

Листинг 2.1.7. Хронометраж одного алгоритма сортировки из данной главы для заданных входных данных

```
public static double time(String alg, Comparable[] a)
{
   Stopwatch timer = new Stopwatch();
   if (alg.equals("Вставки"))        Insertion.sort(a);
   if (alg.equals("Выбор"))          Selection.sort(a);
   if (alg.equals("Шелла"))          Shell.sort(a);
   if (alg.equals("Слияние"))        Merge.sort(a);
   if (alg.equals("Быстрая"))        Quick.sort(a);
   if (alg.equals("Пирамидальная"))  Heap.sort(a);
   return timer.elapsedTime();
}
```

Свойство Г специально сформулировано несколько расплывчато: значение постоянного коэффициента не указано точно, и в нем не сказано о предположении, что стоимости сравнений и перестановок сходны — поэтому оно применимо к широкому диапазону ситуаций. По возможности в подобных заявлениях мы будем стараться четко оформлять существенные аспекты производительности каждого из изучаемых алгоритмов. Как было сказано в главе 1, каждое рассматриваемое нами *свойство* требует научной проверки в конкретной ситуации — возможно, с учетом более точной гипотезы, основанной на соответствующем *утверждении* (математически верном высказывании).

Для практических целей имеется один важный дополнительный шаг: *экспериментальная проверка гипотезы для имеющихся данных*. Рассмотрение этого шага мы отложим до раздела 2.5 и упражнений. В этом случае, если ключи сортировки могут совпадать и/или не являются случайно упорядоченными, свойство Г может и не быть верным. Массив можно специально случайным образом перетасовать с помощью метода `StdRandom.shuffle()`, но для случаев со значительным количеством одинаковых ключей понадобится более тщательный анализ.

Наши рассмотрения анализа алгоритмов должны быть исходными положениями, а не окончательными заключениями. Если вам придет на ум какой-то другой вопрос о производительности алгоритмов, можете исследовать его самостоятельно с помощью средства наподобие `SortCompare`. Множество возможностей для этого содержатся в упражнениях.

Мы не будем продолжать сравнение производительности для сортировки вставками и сортировки выбором, т.к. имеются гораздо более интересные алгоритмы, которые могут работать в сотни, тысячи, а то и миллионы раз быстрее, чем любая из них. Однако знакомство с этими элементарными алгоритмами полезно по нескольким причинам.

- Они помогают сформулировать общие положения.
- Они помогают создать инструменты для замеров производительности.
- В некоторых особых случаях они работают лучше других алгоритмов.
- На их основе можно разрабатывать более эффективные алгоритмы.

По этим причинам мы будем использовать такой же базовый подход и исследовать вначале элементарные алгоритмы для каждой задачи, рассматриваемой в данной книге, а не только для сортировки. Программы наподобие `SortCompare` играют важную роль в этом постепенном подходе к разработке алгоритмов. На каждом шаге мы можем использовать такую программу, чтобы оценить, дает ли ожидаемый выигрыш в производительности новый алгоритм или улучшенная версия известного алгоритма.

Сортировка Шелла

Чтобы продемонстрировать ценность знания свойств элементарных методов сортировки, сейчас мы рассмотрим быстрый алгоритм, который основан на сортировке вставками. Сортировка вставками работает медленно для больших неупорядоченных массивов из-за того, что перестановки в ней выполняются только для соседних элементов, поэтому при каждой перестановке элементы могут перемещаться по массиву лишь на одну позицию. Например, если элемент с наименьшим ключом окажется в конце массива, для его перемещения в надлежащее место потребуются $N-1$ перестановок. *Сортировка Шелла* представляет собой простое расширение сортировки вставками, которое работает быстрее потому, что разрешает обмен далеко расположенных элементов массива и соз-

дает при этом частично упорядоченные массивы, которые затем можно эффективно доупорядочить сортировкой вставками.

Идея состоит в переупорядочении массива таким образом, чтобы каждые h-е элементы (начиная с любого места массива) составляли упорядоченную последовательность. Такой массив называется h-упорядоченным (рис. 2.1.4).

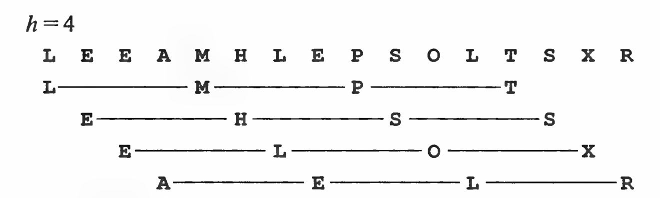


***Рис. 2.1.4.** h-упорядоченная последовательность — это h чередующихся подпоследовательностей сортировки*

Можно сказать и по-другому: h-упорядоченный массив представляет собой h независимо отсортированных перемежающихся подпоследовательностей. h-сортировка для больших значений h позволяет перемещать элементы в массиве на большие расстояния и, таким образом, облегчает выполнение h-сортировки для меньших значений h. Выполнение такой процедуры для любой последовательности значений h, которая заканчивается единицей, приведет к упорядочению массива — это и есть сортировка Шелла (рис. 2.1.5–2.1.7).

Исходные данные	S	H	E	L	L	S	O	R	T	E	X	A	M	P	L	E
13-упорядоченные	P	H	E	L	L	S	O	R	T	E	X	A	M	S	L	E
4-упорядоченные	L	E	E	A	M	H	L	E	P	S	O	L	T	S	X	R
1-упорядоченные	A	E	E	E	H	L	L	L	M	O	P	R	S	S	T	X

***Рис. 2.1.5.** Трассировка сортировки Шелла (содержимое массива после каждого прохода)*

Реализация в алгоритме 2.3 (см. листинг 2.1.8) использует последовательность убывающих значений $\frac{1}{2}(3^k - 1)$, которая начинается с наибольшего шага, не превышающего $N/3$, и заканчивается единицей. Мы будем называть такую последовательность *последовательностью шагов*. Алгоритм 2.3 вычисляет эту последовательность, но можно и хранить ее в массиве.

Листинг 2.1.8. Алгоритм 2.3. Сортировка Шелла

```
public class Shell
{
   public static void sort(Comparable[] a)
   {  // Сортировка a[] по возрастанию.
      int N = a.length;
      int h = 1;
      while (h < N/3) h = 3*h + 1;  // 1, 4, 13, 40, 121, 364, 1093, ...
      while (h >= 1)
      {  // h-сортировка массива.
```

```
            for (int i = h; i < N; i++)
            {  // Вставка a[i] между a[i-h], a[i-2*h], a[i-3*h]... .
               for (int j = i; j >= h && less(a[j], a[j-h]); j -= h)
                  exch(a, j, j-h);
            }
            h = h/3;
         }
      }
   // Реализации less(), exch(), isSorted() и main() см. в листинге 2.1.1.
}
```

Исходные данные	S	H	E	L	L	S	O	R	T	E	X	A	M	P	L	E
13-упорядоченные	**P**	H	E	L	L	S	O	R	T	E	X	A	M	S	L	E
	P	H	E	L	L	S	O	R	T	E	X	A	M	S	**L**	E
	P	H	E	L	L	S	O	R	T	E	X	A	M	S	L	**E**
4-упорядоченные	**L**	H	E	L	P	S	O	R	T	E	X	A	M	S	L	E
	L	H	E	L	P	**S**	O	R	T	E	X	A	M	S	L	E
	L	H	E	L	P	S	**O**	R	T	E	X	A	M	S	L	E
	L	H	E	L	P	S	O	**R**	T	E	X	A	M	S	L	E
	L	H	E	L	P	S	O	R	**T**	E	X	A	M	S	L	E
	L	**E**	E	L	P	H	O	R	T	S	X	A	M	S	L	E
	L	E	E	L	P	H	O	R	T	S	**X**	A	M	S	L	E
	L	E	E	**A**	P	H	O	L	T	S	X	R	M	S	L	E
	L	E	E	A	**M**	H	O	L	P	S	X	R	T	S	L	E
	L	E	E	A	M	H	O	L	P	S	X	R	T	**S**	L	E
	L	E	E	A	M	H	**L**	L	P	S	O	R	T	S	X	E
	L	E	E	A	M	H	L	**E**	P	S	O	L	T	S	X	R
1-упорядоченные	**E**	L	E	A	M	H	L	E	P	S	O	L	T	S	X	R
	E	**E**	L	A	M	H	L	E	P	S	O	L	T	S	X	R
	A	E	E	L	M	H	L	E	P	S	O	L	T	S	X	R
	A	E	E	L	**M**	H	L	E	P	S	O	L	T	S	X	R
	A	E	E	**H**	L	M	L	E	P	S	O	L	T	S	X	R
	A	E	E	H	L	**L**	M	E	P	S	O	L	T	S	X	R
	A	E	E	**E**	H	L	L	M	P	S	O	L	T	S	X	R
	A	E	E	E	H	L	L	M	**P**	S	O	L	T	S	X	R
	A	E	E	E	H	L	L	M	P	**S**	O	L	T	S	X	R
	A	E	E	E	H	L	L	M	**O**	P	S	L	T	S	X	R
	A	E	E	E	H	L	L	**L**	M	O	P	S	T	S	X	R
	A	E	E	E	H	L	L	L	M	O	P	S	**T**	S	X	R
	A	E	E	E	H	L	L	L	M	O	P	S	**S**	T	X	R
	A	E	E	E	H	L	L	L	M	O	P	S	S	T	**X**	R
	A	E	E	E	H	L	L	L	M	O	P	**R**	S	S	T	X
Результат	A	E	E	E	H	L	L	L	M	O	P	R	S	S	T	X

Рис. 2.1.6. Подробная трассировка сортировки Шелла (вставки)

Если изменить сортировку вставками (алгоритм 2.2), чтобы она выполняла h-сортировку массива, и добавить внешний цикл для уменьшения h согласно последовательности шагов, которая начинается с шага, не меньшего определенной части длины массива, и заканчивается единицей, то мы приходим к этой компактной реализации сортировки Шелла.

```
% java SortCompare Шелла Вставки 100000 100
Для 100000 случайных Doubles
   Шелл в 600 раз быстрее, чем Вставки
```

Эффективность сортировки Шелла обусловлена компромиссом между размером и частичной упорядоченностью в подпоследовательностях. Вначале подпоследовательности короткие, а затем они становятся все длиннее, однако они уже частично упорядочены. В обоих случаях для них удобно применять сортировку вставками. Степень частичной упорядоченности подпоследовательностей сильно зависит от последовательности шагов. Оценка производительности сортировки Шелла очень трудна. Вообще говоря, алгоритм 2.3 — единственный метод сортировки из рассматриваемых здесь, производительность которого на случайно упорядоченных массивах не описана точно.

Как определить подходящую последовательность шагов? В общем случае ответ на этот вопрос совсем не прост. Производительность алгоритма зависит не только от количества шагов, но и от арифметической взаимосвязи этих шагов — например, величины их общих делителей и других свойств.

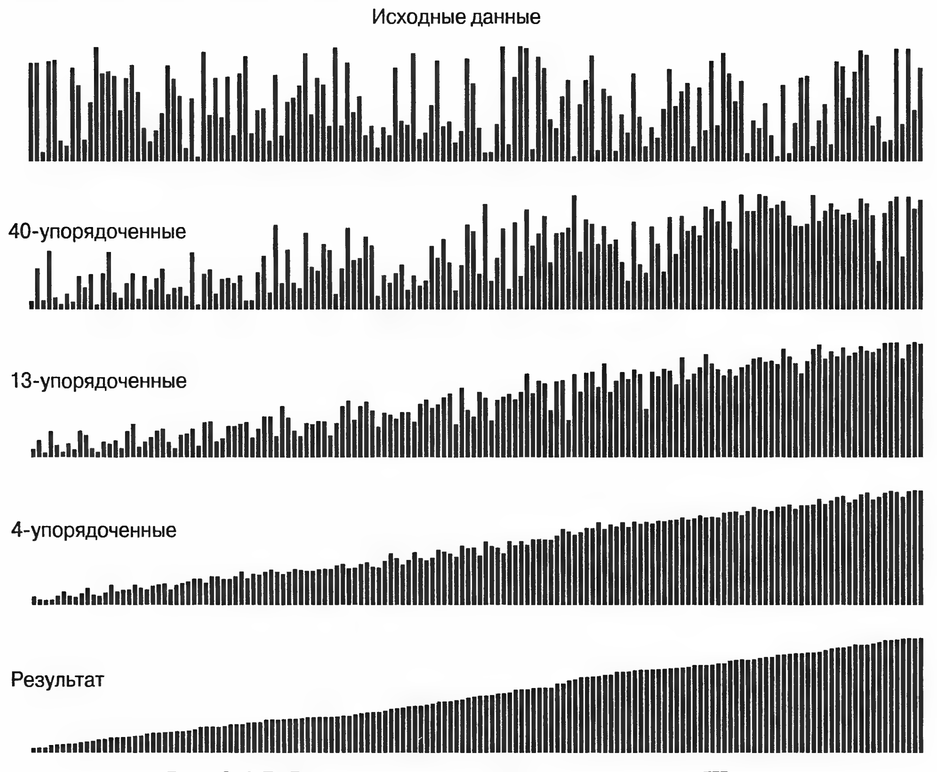


Рис. 2.1.7. Визуальная трассировка сортировки Шелла

В литературе описано множество последовательностей шагов, но, возможно, наилучшая последовательность еще не найдена. Последовательность, используемая в алгоритме 2.3, легко вычислять и применять, а ее производительность почти не хуже более сложных последовательностей, которые (возможно) определяют лучшую производительность в худшем случае. Последовательности шагов, которые ведут себя существенно лучше, возможно, еще ждут своего первооткрывателя.

Сортировка Шелла, в отличие от сортировки выбором и вставками, вполне пригодна даже для больших массивов. Она хорошо работает на массивах с произвольной упорядоченностью (не обязательно со случайной). И весьма трудно подобрать такой массив, для которого сортировка Шелла работала бы медленно для конкретной последовательности шагов.

Как показывает программа `SortCompare`, сортировка Шелла гораздо быстрее сортировки вставками и сортировки выбором, и разница в скорости увеличивается вместе с размером массива. Прежде чем продолжить чтение, попробуйте сравнить на своем компьютере с помощью программы `SortCompare` поведение сортировки Шелла с сортировкой вставками и выбором для размеров массивов, равных степеням 2 (см. упражнение 2.1.27). Вы увидите, что сортировка Шелла позволяет упорядочивать такие объемы данных, на которых застревают более простые алгоритмы. Этот пример является нашей первой практической демонстрацией важного принципа, который пронизывает всю эту книгу: *достижение ускорения, которое позволяет решать задачи, не разрешимые другими средствами — одна из основных причин изучения производительности и проектирования алгоритмов*.

Для изучения характеристик производительности сортировки Шелла необходимы математические выкладки, которые выходят за рамки настоящей книги. Если вы не верите нам на слово, то для начала подумайте, как можно доказать, что *если k-отсортировать h-упорядоченный массив, то он остается h-упорядоченным*. А наиболее важным результатом о производительности алгоритма 2.3 на данный момент является то, что *время выполнения сортировки Шелла не обязательно квадратично*: например, известно, что количество сравнений в алгоритме 2.3 в худшем случае пропорционально $N^{3/2}$. То, что такая простая модификация может преодолеть барьер квадратичности времени выполнения, весьма интересно — ведь это и есть основная цель для многих задач построения алгоритмов.

Не существует математических результатов о среднем количестве сравнений, которое выполняет сортировка Шелла для случайно упорядоченных входных данных. Придуманы последовательности шагов, которые сводят асимптотический рост количества сравнений в худшем случае до $N^{4/3}$, $N^{5/4}$, $N^{6/5}$, ... , но большинство таких результатов имеют в основном академическую ценность, т.к. для практических значений N эти функции трудно отличить одну от другой (и от пропорциональности N).

На практике можно спокойно пользоваться наработками последних научных исследований сортировки Шелла и просто применять последовательность шагов из алгоритма 2.3 (или одну из последовательностей, описанных в упражнениях в конце этого раздела, которые могут повысить производительность на 20–40%). Более того, несложно проверить следующую гипотезу.

Свойство Д. Количество сравнений, которые выполняет сортировка Шелла с шагами 1, 4, 13, 40, 121, 364, ..., ограничено произведением N, количества использованных шагов и небольшого множителя.

Обоснование. В алгоритм 2.3 несложно добавить подсчет сравнений и поделить полученное количество на количество шагов (см. упражнение 2.1.12). Многочисленные эксперименты дают основание предполагать, что среднее количество

сравнений на один шаг может быть равно $N^{1/5}$, но для не очень больших N рост этой функции трудно обнаружить. Данное свойство, по-видимому, не очень зависит от модели входных данных.

Опытные программисты иногда отдают предпочтение сортировке Шелла — за приемлемое время работы даже для довольно больших массивов, за то, что ее несложно закодировать, и за то, что она не требует дополнительной памяти. В последующих разделах мы рассмотрим более эффективные методы, но они работают быстрее всего раза в два (и то не всегда), за исключением очень больших N, и более сложны. Если вам нужно выполнять упорядочение каких-то данных, а системная сортировка почему-то недоступна (например, код предназначен для оборудования или встроенной системы), можно без проблем использовать сортировку Шелла, а потом решить, стоит ли заменять ее более сложным методом.

Вопросы и ответы

Вопрос. **Сортировка — какая-то "игрушечная" задача. Может, на компьютере можно делать и что-то более интересное?**

Ответ. Можно, но многие их этих более интересных вещей *стали возможными* благодаря быстрым алгоритмам сортировки. Многочисленные примеры приведены в разделе 2.5 и в остальной части книги. Сортировка — удобная тема для изучения, т.к. задачу легко понять, а на примере более быстрых алгоритмов научиться хитроумным приемам.

Вопрос. **Зачем придумали так много алгоритмов сортировки?**

Ответ. Одна из причин — производительность многих алгоритмов зависит от входных значений, поэтому различные алгоритмы удобны для различных приложений с различными видами входных данных. Например, сортировка вставками удобна в случае частично упорядоченных массивов или очень маленьких массивов. Важны и другие соображения, например, требования к памяти или возможность обработки одинаковых ключей. К этому вопросу мы еще вернемся в разделе 2.5.

Вопрос. **Зачем нужны крошечные вспомогательные методы `less()` и `exch()`?**

Ответ. Это базовые абстрактные операции, которые необходимы любому алгоритму сортировки, и код гораздо более понятен, если использует эти абстракции. Более того, они делают код непосредственно переносимым в другие условия. Например, значительная часть кода из алгоритмов 2.1 и 2.2 верна и в нескольких других языках программирования. Даже в Java мы можем использовать этот код для сортировки примитивных типов (которые не реализуют интерфейс `Comparable`): нужно просто реализовать метод `less()` с помощью кода `v < w`.

Вопрос. **Когда я запускаю программу `SortCompare`, то получаю каждый раз различные значения (и отличные от приведенных в книге). Почему?**

Ответ. Ну, для начала, вы работаете не на таком компьютере, как мы, а ведь еще могут отличаться операционные системы, среда времени выполнения Java и т.д. Все эти отличия могут привести к небольшим различиям в машинном коде, сгенерированном для алгоритмов. Различия при разных запусках на одном компьютере могут появиться из-за других приложений, которые выполняются на

вашем компьютере, либо из-за других условий. Выполнение очень большого количества попыток должно несколько сгладить этот эффект. Мораль такого тестирования в том, что в настоящее время трудно заметить небольшие отличия в производительности алгоритмов. Вот поэтому (в основном) мы и обращаем внимание на большие отличия!

Упражнения

2.1.1. Покажите в стиле трассировки для алгоритма 2.1, как сортировка выбором сортирует массив `E A S Y Q U E S T I O N`.

2.1.2. Каково максимальное количество перестановок, в которых задействован любой конкретный элемент, во время сортировки выбором? Каково среднее количество перестановок, в которых задействован какой-то элемент?

2.1.3. Приведите пример массива из N элементов, для которого количество неудачных сравнений `a[j] < a[min]` (из-за чего изменяется значение `min`) максимально при работе сортировки выбором (алгоритм 2.1).

2.1.4. Покажите в стиле трассировки для алгоритма 2.2, как сортировка вставками сортирует массив `E A S Y Q U E S T I O N`.

2.1.5. Для каждого из двух условий во внутреннем цикле `for` сортировки вставками (алгоритм 2.2) приведите пример массива из N элементов, где это условие всегда ложно по завершении цикла.

2.1.6. Какой метод работает быстрее для массива со всеми одинаковыми ключами — сортировка выбором или вставками?

2.1.7. Какой метод работает быстрее для массива с обратным упорядочением — сортировка выбором или вставками?

2.1.8. Предположим, что сортировка вставками применяется к случайно упорядоченному массиву, элементы которого принимают одно из трех возможных значений. Будет ли время выполнения линейным, квадратичным или каким-то промежуточным?

2.1.9. Покажите в стиле трассировки для алгоритма 2.3, как сортировка Шелла сортирует массив `E A S Y S H E L L S O R T Q U E S T I O N`.

2.1.10. Почему для *h*-упорядочения в сортировке Шелла не применяется сортировка выбором?

2.1.11. Реализуйте вариант сортировки Шелла, который не вычисляет последовательность шагов, а хранит ее в массиве.

2.1.12. Добавьте в сортировку Шелла вывод на каждом шаге количества сравнений, деленного на размер массива. Напишите клиент тестирования, который проверяет гипотезу, что это количество является небольшой константой, выполнив сортировку массивов случайных значений `Double` с размерами массивов, которые равны возрастающим степеням 10, начиная со 100.

Творческие задачи

2.1.13. *Сортировка колоды.* Поясните, как бы вы упорядочили колоду карт по мастям (в порядке пики, червы, трефы, бубны) и по старшинству в каждой масти, при наличии следующего ограничения. Карты должны лежать в один ряд лицом вниз, и единственной разрешенной операцией является проверка значений двух карт и, возможно, их обмен (не переворачивая).

2.1.14. *Сортировка стеко-очереди.* Поясните, как бы вы упорядочили колоду карт при наличии следующего ограничения. Единственной разрешенной операцией является проверка значений двух верхних карт, возможно, их обмен и помещение вниз колоды.

2.1.15. *Трудоемкий обмен.* Служащему в службе доставки нужно упорядочить ряд больших ящиков по времени их отправки. То есть стоимость сравнения очень низка (достаточно посмотреть на этикетки) по сравнению со стоимостью перестановок (перемещения ящиков). Склад почти заполнен, и дополнительного места хватает только для одного ящика, но не двух. Каким методом сортировки следует воспользоваться служащему?

2.1.16. *Проверка.* Напишите метод `check()`, который вызывает метод `sort()` для заданного массива и возвращает `true`, если массив упорядочен *и* содержит то же множество объектов, что и первоначально, а иначе возвращает `false`. Не считайте, что метод `sort()` может перемещать данные только с помощью операции `exch()`. Можно использовать системный метод `Arrays.sort()` и считать, что он работает корректно.

2.1.17. *Анимация.* Добавьте в программы `Insertion` и `Selection` код для отрисовки содержимого массива в виде вертикальных полосок — как в визуальных трассировках в этом разделе. Полоски должны перерисовываться после каждого прохода, чтобы получился эффект движения, и в конце вывести “отсортированную” картинку, где полоски упорядочены по их высоте. *Подсказка*: используйте клиент вроде приведенного в тексте, который генерирует случайные значения `Double`, добавьте в нужные места кода сортировки вызовы `show()` и реализуйте метод `show()`, который очищает поле чертежа и вычерчивает полоски.

2.1.18. *Визуальная трассировка.* Измените решение предыдущего упражнения так, чтобы `Insertion` и `Selection` выводили визуальные трассировки вроде приведенных в этом разделе. *Совет*: эту задачу можно облегчить разумным применением метода `setYscale()`. *Дополнительное задание*: добавьте код, необходимый для вывода выделений красным и серым цветом.

2.1.19. *Худший случай для сортировки Шелла.* Подберите массив из 100 элементов, который содержит числа от 1 до 100 и для которого сортировка Шелла с шагами `1 4 13 40` выполняет максимальное количество сравнений, которое вы сможете обнаружить.

2.1.20. *Лучший случай для сортировки Шелла.* Какой случай можно считать *лучшим* для сортировки Шелла? Обоснуйте свой ответ.

2.1.21. *Транзакции с возможностью сравнения.* Используя в качестве модели код для типа `Date` (листинг 2.1.3), добавьте в класс `Transaction` (упражнение 1.2.13) реализацию интерфейса `Comparable`, чтобы упорядочивать транзакции по значениям их сумм.

Решение:

```
public class Transaction implements Comparable<Transaction>
{
   ...
   private final double amount;
   ...
   public int compareTo(Transaction that)
   {
      if (this.amount > that.amount) return +1;
      if (this.amount < that.amount) return -1;
      return 0;
   }
   ...
}
```

2.1.22. *Клиент тестирования для сортировки транзакций.*
Напишите класс `SortTransactions` со статическим методом `main()`, который читает из стандартного ввода последовательность транзакций, сортирует их и выводит результат в стандартный вывод (см. упражнение 1.3.17).

Решение:

```
public class SortTransactions
{
   public static Transaction[] readTransactions()
   {  // См. упражнение 1.3.17  }
   public static void main(String[] args)
   {
      Transaction[] transactions = readTransactions();
      Shell.sort(transactions);
      for (Transaction t : transactions)
         StdOut.println(t);
   }
}
```

Эксперименты

2.1.23. *Сортировка колоды.* Попросите нескольких друзей упорядочить колоду карт (см. упражнение 2.1.13). Заметьте, как они это делают, и закодируйте методы, которые они используют.

2.1.24. *Сортировка вставками с маркером конца.* Напишите реализацию сортировки вставками, которая устраняет из внутреннего цикла проверку `j > 0`, поместив вначале наименьший элемент на свое место. Оцените эффективность такого усовершенствования с помощью программы `SortCompare`. *Примечание*: таким способом можно избежать проверки на достижение границы и во многих других случаях. Элемент, который позволяет устранить проверку, называется *маркером* (sentinel).

2.1.25. *Сортировка вставками без перестановок.* Напишите реализацию сортировки вставками, которая перемещает большие элементы на одну позицию вправо с помощью лишь одного обращения к массиву на элемент, а не использует опе-

рацию `exch()`. Оцените эффективность такого усовершенствования с помощью программы `SortCompare`.

2.1.26. *Примитивные типы.* Напишите версию сортировки вставками для упорядочивания массивов значений `int` и сравните ее производительность с реализацией, приведенной в тексте (которая сортирует значения `Integer` и неявно использует преобразования с помощью автоупаковки и autoраспаковки).

2.1.27. *Субквадратичность сортировки Шелла.* Сравните на своем компьютере с помощью программы `SortCompare` производительность сортировки Шелла с производительностью сортировки вставками и сортировки выбором. Используйте размеры массивов, равные возрастающим степеням 2, начиная со 128.

2.1.28. *Равные ключи.* Сформулируйте и проверьте гипотезу о времени выполнения сортировки вставками и сортировки выбором для массивов, которые содержат только два значения ключей. Считайте, что появление обоих ключей равновероятно.

2.1.29. *Шаги сортировки Шелла.* Экспериментально сравните последовательность шагов из алгоритма 2.3 с последовательностью 1, 5, 19, 41, 109, 209, 505, 929, 2161, 3905, 8929, 16001, 36289, 64769, 146305, 260609, ..., которая получается слиянием последовательностей $9 \cdot 4^k - 9 \cdot 2^k + 1$ и $4^k - 3 \cdot 2^k + 1$. См. упражнение 2.1.11.

2.1.30. *Геометрические шаги.* Экспериментально определите значение t, которое приводит к минимальному времени выполнения сортировки Шелла для случайных массивов с последовательностью шагов 1, $\lfloor t \rfloor$, $\lfloor t^2 \rfloor$, $\lfloor t^3 \rfloor$, $\lfloor t^4 \rfloor$, ... для $N = 10^6$. Приведите значения t и последовательности шагов для наилучших найденных вами значений.

Следующие упражнения описывают различные вспомогательные клиенты для оценки методов сортировки. Они задуманы как начальные средства для оценки свойств производительности для случайных данных. Во всех клиентах используйте функцию `time()` (как в программе `SortCompare`), чтобы получать более точные результаты, указывая больше попыток во втором аргументе командной строки. В последующих разделах мы будем возвращаться к этим упражнениям при оценке более сложных методов.

2.1.31. *Тест с удвоением.* Напишите клиент, который выполняет тест с удвоением для алгоритмов сортировки. Начните с `N=1000` и выводите `N`, прогнозированное количество секунд, реальное количество секунд и отношение при удвоении `N`. Проверьте с помощью своей программы, что сортировка вставками и сортировка выбором являются квадратичными при случайных входных данных, и сформулируйте и проверьте гипотезу для сортировки Шелла.

2.1.32. *График времен выполнения.* Напишите клиент, который использует класс `StdDraw` для вычерчивания графиков, выражающих зависимость времени выполнения алгоритма для случайных данных и различных размеров массивов. Можно добавить один или два дополнительных аргумента командной строки. Постарайтесь создать полезный инструмент.

2.1.33. *Распределение.* Напишите клиент, который входит в бесконечный цикл вызовов `sort()` для массивов с размером, задаваемым третьим аргументом командной строки, замеряет время выполнения каждого прогона и использует класс `StdDraw` для вычерчивания графика средних времен выполнения. Должен появиться рисунок *распределения* времен выполнения.

2.1.34. *Угловые случаи.* Напишите клиент, который вызывает метод `sort()` для трудных или патологических случаев, которые могут возникнуть в практических приложениях. Примерами могут быть уже упорядоченные массивы, обратно упорядоченные массивы, массивы со всеми одинаковыми ключами, массивы, содержащие только два различных значения, и массивы размерами 0 и 1.

2.1.35. *Неравномерные распределения.* Напишите клиент, который генерирует тестовые данные, случайно упорядочивая объекты на основе других, не равномерных, распределений, включая следующие:

- гауссово;
- Пуассона;
- геометрическое;
- дискретное (специальный случай, см. упражнение 2.1.28).

Сформулируйте и проверьте гипотезы о влиянии таких данных на производительность алгоритмов из данного раздела.

2.1.36. *Неравномерные данные.* Напишите клиент, который генерирует тестовые *данные*, не являющиеся равномерными, включая следующие:

- половина данных равна 0, другая половина равна 1;
- половина данных равна 0, половина оставшихся равна 1, половина оставшихся равна 2 и т.д.;
- половина данных равна 0, а половина — случайным целочисленным значениям.

Сформулируйте и проверьте гипотезы о влиянии таких данных на производительность алгоритмов из данного раздела.

2.1.37. *Частичное упорядочение.* Напишите клиент, который генерирует частично упорядоченные массивы, включая следующие:

- 95% упорядочены, а последние содержат случайные значения;
- все элементы находятся в пределах 10 позиций от окончательного места в массиве;
- упорядоченные данные, кроме 5% элементов, случайно разбросанных в массиве.

Сформулируйте и проверьте гипотезы о влиянии таких данных на производительность алгоритмов из данного раздела.

2.1.38. *Различные виды элементов.* Напишите клиент, который генерирует массивы элементов различных типов со случайными значениями ключей, включая следующие:

- ключ `String` (не менее 10 символов) и одно значение `double`;
- ключ `double` и десять значений `String` (во всех не менее 10 символов);
- ключ `int` и одно значение `int[20]`.

Сформулируйте и проверьте гипотезы о влиянии таких данных на производительность алгоритмов из данного раздела.

2.2. Сортировка слиянием

Алгоритмы, которые мы рассмотрим в данном разделе, основаны на простой операции *слияния* — объединения двух упорядоченных массивов для получения одного большего упорядоченного массива. Эта операция непосредственно приводит к простому рекурсивному методу сортировки, который называется ***сортировкой слиянием***: для сортировки массива нужно поделить его пополам, отсортировать (рекурсивно) эти половины и слить результаты (рис. 2.2.1). Как мы увидим, одним из наиболее привлекательных свойств сортировки слиянием является гарантированное время сортировки любого массива из N элементов за время, пропорциональное $N\log N$. Основной ее недостаток — требование дополнительной памяти с объемом, пропорциональным N.

Исходные данные	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
Сортировка левой половины	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
Сортировка правой половины	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
Слияние результатов	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Рис. 2.2.1. Принцип работы сортировки слиянием

Абстрактное слияние на месте

Очевидный способ реализации слияния — метод, который сливает два отдельных упорядоченных массива объектов `Comparable` в третий массив. Эту стратегию несложно реализовать: создайте выходной массив нужного размера, а затем последовательно выбирайте из двух входных массивов наименьший оставшийся в них элемент и добавляйте его в выходной массив.

Но при сортировке большого массива придется выполнять большое количество слияний, поэтому стоимость создания нового массива для каждой порции сливаемых данных может оказаться слишком большой. Гораздо привлекательнее иметь метод сортировки на месте, чтобы отсортировать на месте первую половину массива, потом отсортировать на месте вторую половину массива, а затем слить эти половины, перемещая элементы внутри массива и не используя значительного объема дополнительной памяти. Сейчас имеет смысл остановиться на минутку и подумать, как это можно сделать. С виду эту задачу решить совсем нетрудно, но известные решения довольно сложны, особенно в сравнении с теми, которые требуют дополнительной памяти.

Но *абстракция* слияния на месте все-таки полезна. Поэтому мы будем использовать сигнатуру `merge(a, lo, mid, hi)` для обозначения метода, который помещает результат слияния подмассивов `a[lo..mid]` и `a[mid+1..hi]` в единый упорядоченный массив `a[lo..hi]`. Код, приведенный в листинге 2.2.1, реализует этот метод слияния всего лишь в нескольких строках: он копирует все данные во вспомогательный массив, а затем сливает их в исходный массив. Другой способ описан в упражнении 2.2.10.

Листинг 2.2.1. Абстрактное слияние на месте

```
public static void merge(Comparable[] a, int lo, int mid, int hi)
{ // Слияние a[lo..mid] с a[mid+1..hi].
  int i = lo, j = mid+1;
  for (int k = lo; k <= hi; k++)  // Копирование a[lo..hi] в aux[lo..hi].
    aux[k] = a[k];
  for (int k = lo; k <= hi; k++)  // Слияние назад в a[lo..hi].
    if      (i > mid)              a[k] = aux[j++];
    else if (j > hi )              a[k] = aux[i++];
    else if (less(aux[j], aux[i])) a[k] = aux[j++];
    else                           a[k] = aux[i++];
}
```

Для выполнения слияния данный метод сначала копирует данные во вспомогательный массив aux[], а затем сливает их обратно в a[] (рис. 2.2.2). В процессе слияния (второй цикл for) возможны четыре варианта: левая половина закончилась (берем данные из правой), правая половина закончилась (берем данные из левой), текущий ключ из правой половины меньше текущего ключа из левой половины (берем справа) и текущий ключ из правой половины больше или равен текущему ключу из левой половины (берем слева).

	a[]												aux[]									
k	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**i**	**j**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Входные данные	E	E	G	M	R	A	C	E	R	T			-	-	-	-	-	-	-	-	-	-
Копия	E	E	G	M	R	A	C	E	R	T			E	E	G	M	R	A	C	E	R	T
											0	5										
0	**A**										0	6	E	E	G	M	R	**A**	C	E	R	T
1	A	**C**									0	7	E	E	G	M	R		**C**	E	R	T
2	A	C	**E**								1	7	**E**	E	G	M	R			E	R	T
3	A	C	E	**E**							2	7		**E**	G	M	R			E	R	T
4	A	C	E	E	**E**						2	8			G	M	R			**E**	R	T
5	A	C	E	E	E	**G**					3	8			**G**	M	R				R	T
6	A	C	E	E	E	G	**M**				4	8				**M**	R				R	T
7	A	C	E	E	E	G	M	**R**			5	8					**R**				R	T
8	A	C	E	E	E	G	M	R	**R**		5	9									**R**	T
9	A	C	E	E	E	G	M	R	R	**T**	6	10										**T**
Результат слияния	A	C	E	E	E	G	M	R	R	T												

Рис. 2.2.2. Трассировка абстрактного слияния на месте

Нисходящая сортировка слиянием

Алгоритм 2.4 (см. листинг 2.2.2) представляет собой рекурсивную реализацию сортировки слиянием, основанную на этом абстрактном слиянии на месте. Это один из наиболее известных примеров применения парадигмы *разделяй и властвуй* для построения эффективных алгоритмов. На данном рекурсивном коде основано индуктивное доказательство, что алгоритм действительно упорядочивает массив: если он сортирует два подмассива, то он сортирует и весь массив, сливая их вместе.

Листинг 2.2.2. Нисходящая сортировка слиянием

```
public class Merge
{
   private static Comparable[] aux;      // Вспомогательный массив для слияний.
   public static void sort(Comparable[] a)
   {
      aux = new Comparable[a.length];    // Память выделяется один раз.
      sort(a, 0, a.length - 1);
   }
   private static void sort(Comparable[] a, int lo, int hi)
   {  // Сортировка a[lo..hi].
      if (hi <= lo) return;
      int mid = lo + (hi - lo)/2;
      sort(a, lo, mid);          // Сортировка левой половины.
      sort(a, mid+1, hi);        // Сортировка правой половины.
      merge(a, lo, mid, hi);     // Слияние результатов (листинг 2.2.1).
   }
}
```

Для упорядочения подмассива a[lo..hi] мы делим его на две части: a[lo..mid] и a[mid+1..hi], сортируем их независимо (рекурсивными вызовами) и сливаем упорядоченные подмассивы для получения результата (рис. 2.2.3).

lo hi	a[] 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a,**0**,0,**1**)	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a,**2**,2,**3**)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a,**0**,1,**3**)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a,**4**,4,**5**)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a,**6**,6,**7**)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a,**4**,5,**7**)	E	G	M	R	E	O	R	S	T	E	X	A	M	P	L	E
merge(a,**0**,3,**7**)	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
merge(a,**8**,8,**9**)	E	E	G	M	O	R	R	S	E	T	X	A	M	P	L	E
merge(a,**10**,10,**11**)	E	E	G	M	O	R	R	S	E	T	A	X	M	P	L	E
merge(a,**8**,9,**11**)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
merge(a,**12**,12,**13**)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
merge(a,**14**,14,**15**)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	E	L
merge(a,**12**,13,**15**)	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
merge(a,**8**,11,**15**)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge(a,**0**,7,**15**)	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Рис. 2.2.3. *Трассировка результатов слияния при выполнении нисходящей сортировки слиянием*

Чтобы разобраться, как работает сортировка слиянием, полезно внимательно рассмотреть динамику вызовов методов, показанную на рис. 2.2.4. Чтобы упорядочить содержимое массива a[0..15], метод sort() вызывает себя для сортировки a[0..7], затем для сортировки a[0..3] и a[0..1] и, наконец, после вызовов с a[0] и a[1] приступает к их слиянию (для краткости мы опустили вызовы для простейших вызовов с одним элементом). После этого выполняются слияния a[2] с a[3], a[0..1] с a[2..3] и т.д.

Из этой трассировки видно, что код сортировки просто организует последовательность вызовов метода `merge()`. Мы еще вернемся к этому наблюдению ниже в настоящем разделе. Рекурсивный код дает также основания для анализа времени выполнения сортировки слиянием. Мы проведем этот анализ максимально подробно, т.к. сортировка слиянием является прототипом общего принципа построения алгоритмов "разделяй и властвуй".

Утверждение Е. При упорядочении любого массива длиной N нисходящая сортировка слиянием использует от $\frac{1}{2}N\lg N$ до $N\lg N$ сравнений.

Доказательство. Пусть $C(N)$ — количество сравнений, необходимое для упорядочения массива длиной N. В крайних случаях $C(0) = C(1) = 0$, а для $N > 0$ можно записать рекуррентное отношение для верхней границы, которое непосредственно соответствует рекурсивному методу `sort()`:

$$C(N) \le C(\lfloor N/2 \rfloor) + C(\lceil N/2 \rceil) + N$$

Первое слагаемое в правой части — количество сравнений для упорядочения левой половины массива, второе — количество сравнений для упорядочения правой половины, а третье — количество сравнений при сортировке. Нижняя граница

$$C(N) \ge C(\lfloor N/2 \rfloor) + C(\lceil N/2 \rceil) + \lfloor N/2 \rfloor$$

следует из того, что количество сравнений при слиянии не меньше $\lfloor N/2 \rfloor$.

Сортировка левой половины

```
sort(a, 0, 15)
  sort(a, 0, 7)
    sort(a, 0, 3)
      sort(a, 0, 1)
        merge(a, 0, 0, 1)
      sort(a, 2, 3)
        merge(a, 2, 2, 3)
      merge(a, 0, 1, 3)
    sort(a, 4, 7)
      sort(a, 4, 5)
        merge(a, 4, 4, 5)
      sort(a, 6, 7)
        merge(a, 6, 6, 7)
      merge(a, 4, 5, 7)
    merge(a, 0, 3, 7)
  sort(a, 8, 15)
    sort(a, 8, 11)
      sort(a, 8, 9)
        merge(a, 8, 8, 9)
      sort(a, 10, 11)
        merge(a, 10, 10, 11)
      merge(a, 8, 9, 11)
    sort(a, 12, 15)
      sort(a, 12, 13)
        merge(a, 12, 12, 13)
      sort(a, 14, 15)
        merge(a, 14, 14, 15)
      merge(a, 12, 13, 15)
    merge(a, 8, 11, 15)
  merge(a, 0, 7, 15)
```

Сортировка правой половины

Слияние результатов

Рис. 2.2.4. Трассировка вызовов нисходящей сортировки слиянием

Мы выведем точное решение рекуррентного уравнения для случая точного равенства и N, равного степени 2 (т.е. $N = 2^n$). Во-первых, поскольку $\lfloor N/2 \rfloor = \lceil N/2 \rceil = 2^{n-1}$, то

$$C(2^n) = 2C(2^{n-1}) + 2^n$$

Поделив обе части этого равенства на 2^n, получим

$$C(2^n)/2^n = C(2^{n-1})/2^{n-1} + 1$$

Применение этого же равенства к первому слагаемому в правой части дает

$$C(2^n)/2^n = C(2^{n-2})/2^{n-2} + 1 + 1$$

И, повторив предыдущий шаг еще $n-1$ раз, получим

$$C(2^n)/2^n = C(2^0)/2^0 + n$$

откуда, после умножения обеих частей на 2^n, следует решение:

$$C(N) = C(2^n) = n2^n = N\lg N$$

Точные решения для произвольных N более сложны, но аналогичные рассуждения в отношении неравенств пригодны и для границ количества сравнений при произвольных значениях N. Это доказательство верно вне зависимости от входных значений и их порядка.

Для понимания утверждения Е удобно рассмотреть дерево, приведенное на рис. 2.2.5 — в нем каждый узел означает подмассив, для которого метод `sort()` выполняет операцию `merge()`. Это дерево состоит точно из n уровней. На k-м сверху уровне ($k = 0, ..., n-1$) имеются 2^k подмассивов, каждый длиной 2^{n-k}, и каждый требует для слияния 2^{n-k} сравнений. Поэтому для каждого из n уровней общая стоимость равна 2^n, а для всего дерева необходимо $n2^n = N\lg N$ сравнений.

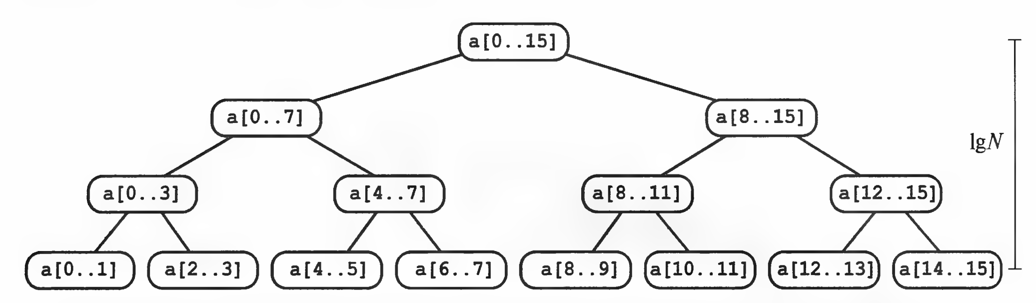


Рис. 2.2.5. *Дерево зависимостей подмассивов для сортировки слиянием при $N = 16$*

Утверждение Ж. Для упорядочения массива длиной N нисходящая сортировка слиянием использует не более $6N\lg N$ обращений к массиву.

Доказательство. Каждое слияние обращается к массиву максимум $6N$ раз: $2N$ для копирования, $2N$ — для записи назад и не более $2N$ — для сравнений. Результат следует из таких же рассуждений, как и для утверждения Е.

Из утверждений Е и Ж следует, что для сортировки слиянием можно ожидать время выполнения, пропорциональное $N\lg N$. Так что по сравнению с элементарными методами из раздела 2.1 мы переходим на другой уровень: теперь мы можем сортировать очень большие массивы, затрачивая лишь в логарифмическое количество раз больше времени, чем на просмотр каждого отдельного элемента. Сортировка слиянием позволяет сортировать многие миллионы элементов, но это невозможно с помощью сортировки вставками или выбором. Основным недостатком сортировки слиянием можно считать требование дополнительной памяти объемом, пропорциональным N, которая нужна для вспомогательного массива. Если память является дорогостоящим ресурсом, то необходим другой метод. Но, с другой стороны, можно существенно сократить время выполнения сортировки слиянием, подобрав для реализации специальные модификации.

Использование сортировки вставками для маленьких подмассивов

Большинство рекурсивных алгоритмов можно улучшить, обрабатывая небольшие случаи специальным образом. Рекурсия *гарантирует*, что небольшие случаи встречаются очень часто, поэтому усовершенствование их обработки приводит к усовершенствованию всего алгоритма. В случае сортировки мы знаем, что сортировка вставками (или сортировка выбором) проста и поэтому может работать быстрее для маленьких подмас-

сивов, чем сортировка слиянием. Как обычно, понять функционирование сортировки слиянием помогает визуальная трассировка. На рис. 2.2.6 демонстрируется работа реализации сортировки слиянием с отсечкой для небольших подмассивов. Переключение на сортировку вставками для небольших подмассивов (скажем, длиной 15 или меньше) может улучшить время выполнения типичной реализации сортировки слиянием на 10–15% (см. упражнение 2.2.23).

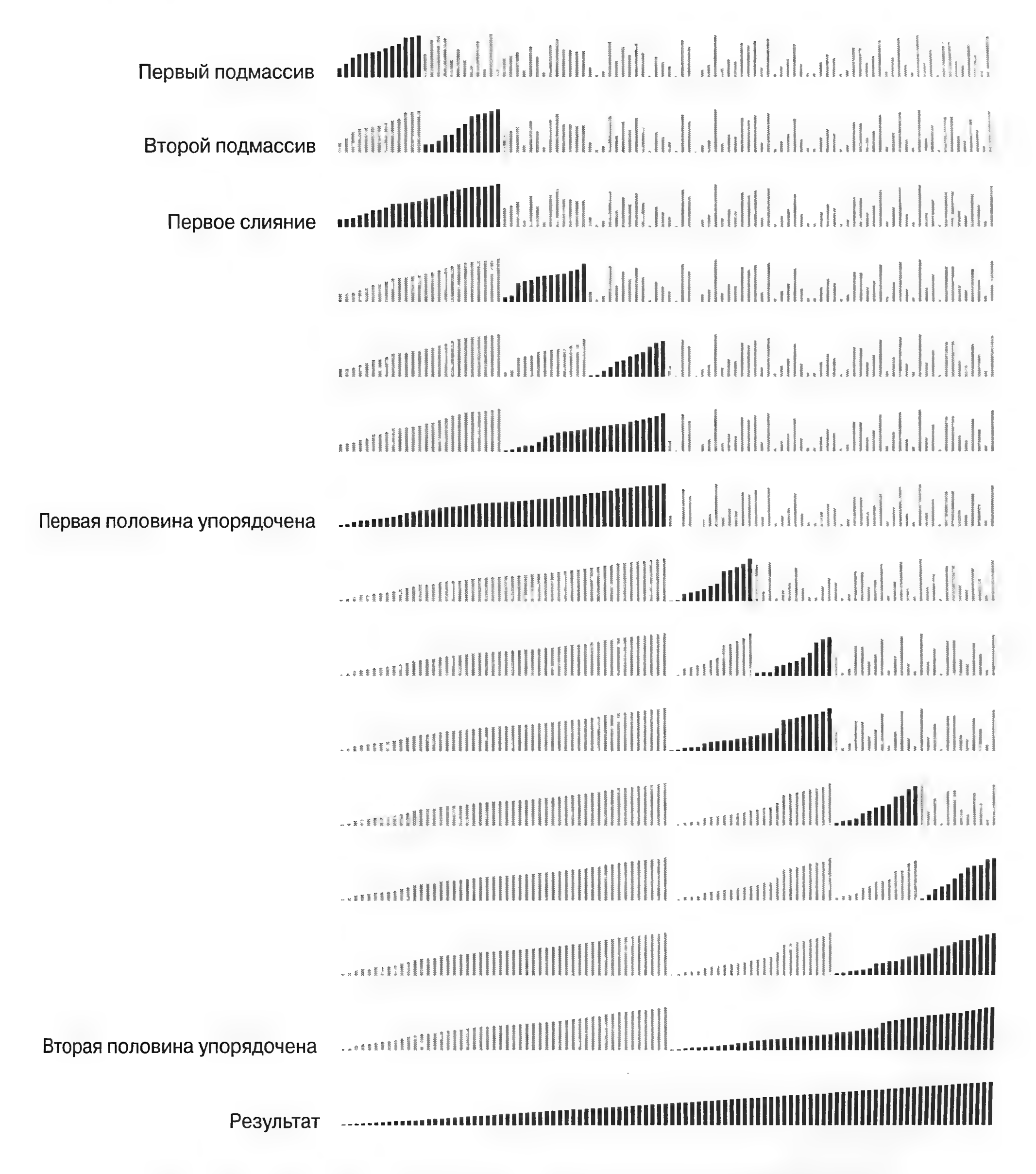


Рис. 2.2.6. *Визуальная трассировка нисходящей сортировки слиянием с отсечкой небольших подмассивов*

9 Зак. 3239

Проверка массива на изначальную упорядоченность

Время выполнения для уже упорядоченных массивов можно снизить до линейного, введя проверку на пропуск вызова `merge()`, если `a[mid]` меньше или равно `a[mid+1]`. При этом все рекурсивные вызовы также выполняются, но время выполнения для упорядоченных подмассивов уже линейно (см. упражнение 2.2.8).

Отказ от копирования во вспомогательный массив

Можно устранить время (но не память), необходимое для копирования во вспомогательный массив при слиянии. Для этого нужно воспользоваться двумя разновидностями метода сортировки: одна берет данные из указанного массива и помещает упорядоченные выходные данные во вспомогательный массив, а другая берет данные из вспомогательного массива и помещает упорядоченные выходные данные в исходный массив. С помощью этого приема и небольших рекурсивных хитростей можно так упорядочить рекурсивные вызовы, что входной и вспомогательный массивы будут меняться ролями на каждом уровне (см. упражнение 2.2.11).

Здесь уместно повторить то, о чем уже было сказано в главе 1, но что легко забыть. Локально каждый алгоритм в этой книге рассматривается как критический для какого-то применения. А глобально мы пытаемся нащупать общие выводы, чтобы рекомендовать какой-то подход. Наши описания таких усовершенствований не обязательно означают рекомендацию реализовывать их во всех ситуациях — скорее, это рекомендация не делать абсолютных выводов о производительности на основе первых реализаций. При рассмотрении новой задачи лучше всего использовать самую простую реализацию, с которой у вас не будет никаких проблем, а затем усовершенствовать ее, если она станет узким местом в приложении. Обычно не стоит тратить время на улучшения, которые уменьшают время выполнения только на небольшой постоянный множитель. Поэтому необходимо экспериментально проверять эффективность конкретных усовершенствований, как мы постоянно указываем в упражнениях.

В случае сортировки слиянием три перечисленных выше улучшения несложно реализовать, и они вполне пригодны, если в конкретной ситуации оптимален метод слияния — например, в ситуациях, описанных в конце данной главы.

Восходящая сортировка слиянием

Рекурсивная реализация сортировки слиянием представляет собой прототип принципа *разделяй и властвуй*, где для решения большая задача делится на части, затем решаются меньшие подзадачи, а на основе полученных решений выводится решение и всей исходной задачи (рис. 2.2.7). Даже если мы обдумываем слияние двух больших подмассивов, на самом деле бо́льшая часть времени уходит на слияния маленьких подмассивов. Но слияния можно организовать и другим способом: сначала выполнить их для маленьких подмассивов, на следующем проходе попарно слить эти подмассивы и продолжать так, пока слияние не охватит весь массив. Код для такого метода даже короче, чем стандартная рекурсивная реализация. Сначала выполняется проход слияний 1 с 1 (отдельные элементы считаются подмассивами размера 1), потом проход слияний 2 с 2 (сливаются подмассивы размера 2 и получаются подмассивы размера 4), затем проход слияний 4 с 4 и т.д. В последнем слиянии на каждом проходе второй подмассив может быть короче первого (что не представляет сложности для метода `merge()`), но в остальных случаях все слияния обрабатывают подмассивы одинакового размера и удваивают размер упорядоченных подмассивов для следующего прохода.

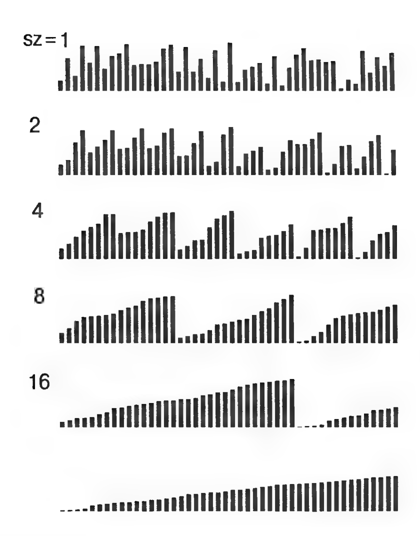


Рис. 2.2.7. Визуальная трассировка восходящей сортировки слиянием

Листинг 2.2.3. Восходящая сортировка слиянием

```
public class MergeBU
{
  private static Comparable[] aux;     // вспомогательный массив для слияний
  // Код merge() см. в листинге 2.2.1.
  public static void sort(Comparable[] a)
  { // Выполнение lgN проходов попарных слияний.
    int N = a.length;
    aux = new Comparable[N];
    for (int sz = 1; sz < N; sz = sz+sz)        // sz — размер подмассива
      for (int lo = 0; lo < N-sz; lo += sz+sz) // lo — индекс в подмассиве
        merge(a, lo, lo+sz-1, Math.min(lo+sz+sz-1, N-1));
  }
}
```

Восходящая сортировка слиянием состоит из последовательности проходов по всему массиву, на каждом их которых выполняются слияния sz с sz. Сначала sz = 1, а на каждом проходе это значение удваивается. Последний подмассив имеет размер sz только в том случае, если размер массива кратен удвоенному sz (иначе он меньше sz) (рис. 2.2.8).

Утверждение 3. Для упорядочения массива длиной N восходящая сортировка слиянием использует от $\frac{1}{2}N\lg N$ до $N\lg N$ сравнений и не более $6N\lg N$ обращений к массиву.

Доказательство. Количество проходов по массиву в точности равно $\lfloor \lg N \rfloor$ (а это значение в точности равно такому n, что $2^n \le N < 2^{n+1}$). На каждом проходе количество обращений к массиву в точности равно $6N$, а количество сравнений не больше N и не меньше $N/2$.

	a[i]															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
sz=1	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 0, 0, 1)	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 2, 2, 3)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 4, 4, 5)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 6, 6, 7)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 8, 8, 9)	E	M	G	R	E	S	O	R	E	T	X	A	M	P	L	E
merge(a,10,10,11)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
merge(a,12,12,13)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
merge(a,14,14,15)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	E	L
sz=2																
merge(a, 0, 1, 3)	E	G	M	R	E	S	O	R	E	T	A	X	M	P	E	L
merge(a, 4, 5, 7)	E	G	M	R	E	O	R	S	E	T	A	X	M	P	E	L
merge(a, 8, 9,11)	E	G	M	R	E	O	R	S	A	E	T	X	M	P	E	L
merge(a,12,13,15)	E	G	M	R	E	O	R	S	A	E	T	X	E	L	M	P
sz=4																
merge(a,0, 3, 7)	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
merge(a,8,11,15)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
sz=8																
merge(a,0,7,15)	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Рис. 2.2.8. *Трассировка результатов слияния при выполнении восходящей сортировки слиянием*

Если длина массива равна степени 2, то нисходящая и восходящая сортировки слиянием выполняют совершенно одинаковые сравнения и обращения к элементам массива, только в разном порядке. Если длина массива не равна степени 2, то последовательности сравнений и обращений к массиву для этих двух алгоритмов будут различными (см. упражнение 2.2.5).

Восходящая сортировка слиянием удобна при организации данных в виде *связных списков*. Сначала сортируемый массив рассматривается как последовательность подсписков длиной 1, потом проход по массиву создает связанные между собой упорядоченные подмассивы длиной 2, затем длиной 4 и т.д. Этот метод переупорядочивает ссылки и поэтому может сортировать список *на месте* (без создания новых узлов списка).

И нисходящий, и восходящий подходы к реализации алгоритмов типа "разделяй и властвуй" интуитивно понятны. Из реализаций сортировки слиянием можно сделать следующий вывод: когда рассматривается алгоритм, основанный на одном из этих подходов, полезно рассмотреть и другой подход. Задачу иногда удобнее решать, разбивая ее на меньшие задачи (с рекурсивным их решением), как в методе `Merge.sort()`, а иногда — объединяя маленькие решения в большие, как в методе `MergeBU.sort()`.

Сложность сортировки

Сортировка слиянием важна, в частности, тем, что она используется как базис для доказательства фундаментального результата в области *вычислительной сложности*, который помогает понять сложность сортировки вообще. Как правило, вычислительная сложность играет важную роль в проектировании алгоритмов, и этот результат непосредственно влияет на проектирование алгоритмов сортировки, поэтому сейчас мы займемся этим вплотную.

Первое, что нужно сделать при изучении сложности — определиться с моделью вычислений. Обычно исследователи стараются найти простейшую модель, которая все-таки соотносится с задачей. В случае сортировки мы рассматриваем класс алгоритмов, *основанных на сравнениях*, которые принимают решения об элементах только на основе сравнения их ключей. Основанный на сравнениях алгоритм может выполнить произвольный объем вычислений между сравнениями, но он не может получить какую-либо информацию о ключе кроме его сравнения с другим ключом. Поскольку мы ограничиваемся API-интерфейсом `Comparable`, все алгоритмы в настоящей главе находятся в этом классе (с игнорированием стоимости обращений к массиву) — как, впрочем, и многие другие алгоритмы, которые мы можем себе представить. В главе 5 мы рассмотрим алгоритмы, которые не опираются на свойство `Comparable` элементов.

Утверждение И. Ни один алгоритм сортировки, основанный на сравнениях, не может гарантированно упорядочить N элементов, выполнив менее $\lg(N!) \sim N\lg N$ сравнений.

Доказательство. Первым делом, мы будем считать, что все ключи различны, т.к. такую сортировку должен уметь выполнить любой алгоритм. Тогда для описания последовательности сравнений можно использовать бинарное дерево. Каждый *узел* в этом дереве представляет собой либо *лист* ($i_0 i_1 i_2 \ldots i_{N-1}$), который означает, что сортировка завершена и исходные данные находятся в порядке $a[i_0]$, $a[i_1]$, ..., $a[i_{N-1}]$, либо *внутренний узел* ($i{:}j$) — такой узел соответствует операции сравнения элементов `a[i]` и `a[j]`, и его левое поддерево соответствует последовательности сравнений в случае, если `a[i]` меньше `a[j]`, а правое — случаю, если `a[i]` больше `a[j]`. Путь от корня до любого листа соответствует последовательности сравнений, выполненных алгоритмом для формирования упорядоченности, сформулированной в листе. Например, на рис. 2.2.9 показано дерево сравнений для N=3.

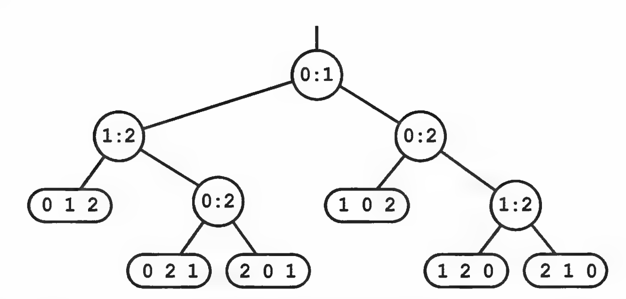


Рис. 2.2.9. *Дерево сравнений при сортировке трех элементов*

Мы никогда не будем явно создавать такие деревья — это просто математический аппарат для описания сравнений, используемых любым алгоритмом.

Первое заключение, важное для доказательства: дерево должно содержать не менее $N!$ листьев, т.к. существуют $N!$ различных перестановок из N различных ключей. Если листьев меньше $N!$, то, значит, какие-то перестановки ими не охвачены, и алгоритм не сможет работать для таких сочетаний ключей.

Количество внутренних узлов на пути от корня к любому листу дерева равно количеству сравнений, используемых алгоритмом для некоторых входных данных.

Нас интересует длина самого длинного такого пути в дереве (*высота* дерева), поскольку она равна количеству сравнений, выполняемых алгоритмом в худшем случае. Одно из основных комбинаторных свойств бинарных деревьев гласит, что дерево высотой h не может иметь более 2^h листьев: дерево высотой h с максимальным количеством листьев полностью сбалансировано, т.е. *полно*. Пример для $h = 4$ приведен на рис. 2.2.10.

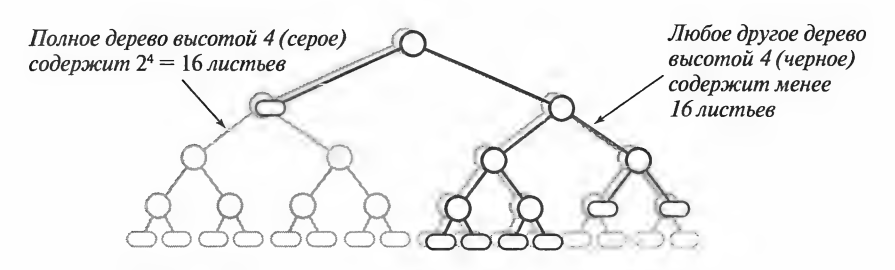


***Рис. 2.2.10.** Полное и произвольное деревья высотой 4*

Из последних двух абзацев следует, что любой алгоритм сортировки, основанный на сравнениях, соответствует дереву сравнений высотой h, такой, что

$$N! \leq \textit{количество листьев} \leq 2^h$$

Это проиллюстрировано на рис. 2.2.11.

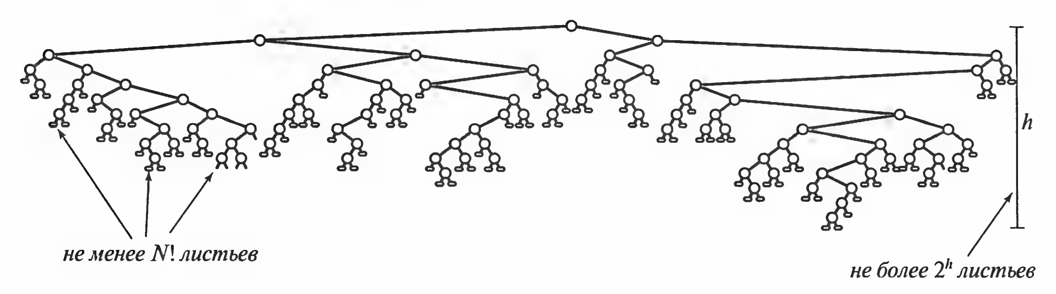


***Рис. 2.2.11.** Количество листьев в дереве сравнений*

Значение h в точности равно количеству сравнений в худшем случае, поэтому можно взять логарифм (по основанию 2) от крайних частей этой формулы и получить, что количество сравнений, используемых любым алгоритмом, не может быть меньше $\lg(N!)$. Приблизительное значение $\lg(N!) \sim N\lg N$ следует из аппроксимации Стирлинга для функции факториала (см. табл. 1.4.5).

Этот результат — указание, чего следует ожидать при проектировании алгоритма сортировки. Например, не имея такого результата, кто-то мог бы долго искать алгоритм сортировки на основе сравнений, который использует в худшем случае вдвое меньше сравнений, чем сортировка слиянием. Нижняя граница в утверждении И показывает, что подобные усилия не увенчаются успехом: *такой алгоритм невозможен*. Это очень сильное высказывание, применимое к любому мыслимому алгоритму на основе сравнений.

Утверждение З гласит, что количество сравнений, выполняемых сортировкой слиянием в худшем случае, равно $\sim N\lg N$. Этот результат — *верхняя граница* сложности задачи сортировки, в том смысле, что лучшему алгоритму пришлось бы гарантировать выполнение меньшего количества сравнений. Утверждение И гласит, что ни один алгоритм сортировки не может гарантировать выполнение меньше $\sim N\lg N$ сравнений. Это *нижняя граница* сложности задачи сортировки в том смысле, что даже самый наилучший алгоритм должен использовать не менее такого количества сравнений в худшем случае. Давайте посмотри, что означает все это вместе.

Утверждение К. Сортировка слиянием является асимптотически оптимальным алгоритмом сортировки на основе сравнений.

Доказательство. Это утверждение в точности означает, что *количество сравнений, выполняемых сортировкой слиянием в худшем случае, и минимальное количество сравнений, которое может гарантировать любой алгоритм сортировки на основе сравнений, равны* $\sim N\lg N$. Эти факты уже установлены в утверждениях З и И.

Здесь важно понимать, что, как и в случае с моделью вычислений, необходимо четко определить, *что* мы понимаем под оптимальным алгоритмом. Например, можно ужесточить определение оптимальности и требовать, чтобы оптимальный алгоритм сортировки выполнял *в точности* $\lg(N!)$ сравнений. Мы не будем этого делать, т.к. при больших N мы не сможем заметить разницу между таким алгоритмом и (к примеру) сортировкой слиянием. Либо можно расширить определение оптимальности, чтобы ему соответствовал любой алгоритм сортировки, для которого количество сравнений в худшем случае равно $N\lg N$ *с некоторым постоянным коэффициентом*. Мы не будем делать и этого, поскольку разница между таким алгоритмом и сортировкой слиянием будет хорошо заметна при больших N.

Вопрос вычислительной сложности может показаться несколько абстрактным, но относительно фундаментальных исследований сложности, присущей решению вычислительных задач, сомнения обычно не возникают. Более того, в случае, когда приходится рассматривать этот вопрос, именно он помогает разработке хорошего ПО. Во-первых, хорошие верхние границы позволяют проектировщикам ПО получить гарантии производительности: описано много случаев, когда поиск причин низкой производительности приводил к обнаружению квадратичной сортировки вместо линейно-логарифмической. Во-вторых, хорошо определенные нижние границы позволяют не тратить силы на поиски повышения производительности, которое в принципе недоступно.

Однако оптимальность сортировки слиянием еще не означает, что рассказ о сортировке завершен и нет смысла рассматривать другие методы. Дело в том, что теория, изложенная в данном разделе, имеет ряд ограничений, например:

- сортировка слиянием не оптимальна в смысле использования памяти;
- худший случай не обязательно возникает на практике;
- могут быть важны и другие операции, отличные от сравнений (например, обращения к массиву);
- некоторые виды данных можно упорядочивать, *вообще* не выполняя сравнений.

Поэтому в настоящей книге мы рассмотрим и несколько других методов сортировки.

Вопросы и ответы

Вопрос. **Сортировка слиянием быстрее, чем сортировка Шелла?**

Ответ. На практике времена их выполнения отличаются друг от друга на небольшой постоянный множитель (если в сортировке Шелла применяется тщательно проверенная последовательность шагов, вроде приведенной в алгоритме 2.3), поэтому относительная производительность зависит от реализаций.

```
% java SortCompare Слияние Шелла 100000
Для 100000 случайных Double
   Слияние в 1.2 раза быстрее, чем Шелла
```

Пока никто не смог теоретически доказать, что сортировка Шелла является линейно-логарифмической для случайных данных, так что существует шанс, что асимптотический рост средней производительности сортировки Шелла выше. Такой разрыв доказан для производительности в худшем случае, но он не имеет практического значения.

Вопрос. **Почему бы не сделать массив `aux[]` локальным в методе `merge()`?**

Ответ. Чтобы не было излишних расходов на создание массива при каждом слиянии, даже небольшого размера. Иначе эти затраты доминировали бы во времени выполнения сортировки слиянием (см. упражнение 2.2.26). Более правильное решение (которое мы не рассматривали, чтобы не загромождать код) — сделать массив `aux[]` локальным в методе `sort()` и передавать его в качестве аргумента в `merge()` (см. упражнение 2.2.9).

Вопрос. **Как ведет себя сортировка слиянием при наличии в массиве одинаковых значений?**

Ответ. Если все элементы имеют одинаковое значение, время выполнения линейно (при наличии дополнительной проверки и пропуска слияния, если массив уже упорядочен). Но если имеется несколько повторяющихся значений, то такой выигрыш в производительности достигается не всегда. Например, предположим, что входной массив содержит N элементов с одним значением в нечетных позициях и N элементов с другим значением в четных позициях. Для такого массива время выполнения является линейно-логарифмическим (в соответствии с рекуррентными соотношениями для элементов со всеми различными значениями), а не линейным.

Упражнения

2.2.1. Приведите трассировку, в стиле показанной на рис. 2.2.2, для слияния ключей `A E Q S U Y E I N O S` с помощью абстрактного метода `merge()` слияния на месте.

2.2.2. Приведите трассировки, в стиле показанной на рис. 2.2.3, для сортировки ключей `E A S Y Q U E S T I O N` с помощью нисходящей сортировки слиянием.

2.2.3. Выполните упражнение 2.2.2 для восходящей сортировки слиянием.

2.2.4. Правда ли, что абстрактное слияние на месте генерирует правильные выходные данные тогда и только тогда, когда два входных подмассива упорядочены? Обоснуйте свой ответ или приведите контрпример.

2.2.5. Приведите последовательность размеров подмассивов в слияниях, выполняемых алгоритмами нисходящей и восходящей сортировки слиянием для $N = 39$.

2.2.6. Напишите программу для вычисления точного количества обращений к массиву, выполняемых нисходящей и восходящей сортировками слиянием. С помощью этой программы начертите график для N от 1 до 512 и сравните полученные значения с верхней границей $6N\lg N$.

2.2.7. Покажите, что количество сравнений, которое использует сортировка слиянием, монотонно увеличивается ($C(N+1) > C(N)$ для всех $N > 0$).

2.2.8. Предположим, что в алгоритм 2.4 добавлен пропуск вызова `merge()`, если `a[mid] <= a[mid+1]`. Докажите, что количество сравнений, используемых при сортировке слиянием уже упорядоченного массива, линейно.

2.2.9. В библиотечном ПО не рекомендуется задействовать статические массивы наподобие `aux[]`, т.к. данный класс могут одновременно использовать несколько клиентов. Приведите реализацию `Merge`, в которой не применяется статический массив. *Не* делайте массив `aux[]` локальным в методе `merge()` (см. "Вопросы и ответы" в данном разделе). *Совет*: передавайте вспомогательный массив в качестве аргумента в рекурсивный метод `sort()`.

Творческие задачи

2.2.10. *Более быстрое слияние.* Реализуйте версию метода `merge()`, которая копирует вторую половину массива `a[]` в массив `aux[]` в *убывающем порядке*, а затем выполняет слияние назад в `a[]`. Это изменение позволяет не использовать во внутреннем цикле код проверки, не закончилась ли каждая половина. *Примечание*: такая сортировка работает неустойчиво (см. подраздел "Устойчивость" в разделе 2.5).

2.2.11. *Усовершенствования.* Реализуйте три усовершенствования для сортировки слиянием, которые описаны в данном разделе после утверждения Ж: добавьте отсечку для небольших подмассивов, проверяйте упорядоченность входного массива и не выполняйте копирование, переключая аргументы в рекурсивном коде.

2.2.12. *Сублинейный объем дополнительной памяти.* Разработайте реализацию слияния, которое снижает потребность в дополнительной памяти до $\max(M, N/M)$ с помощью следующего приема. Разбейте массив на N/M блоков размером M (для простоты предположим, что N кратно M). Затем, (1) считая блоки элементами с ключом сортировки, равным ключу первого элемента в блоке, отсортируйте их с помощью сортировки выбором; (2) выполните проход по массиву, сливая первый блок со вторым, потом второй с третьим и т.д.

2.2.13. *Нижняя граница для среднего случая.* Докажите, что *ожидаемое* количество сравнений, которое выполняет любой алгоритм сортировки на основе сравнений, должно быть не менее $\sim N\lg N$ (если все возможные упорядочения входных данных равновероятны). *Подсказка*: ожидаемое количество сравнений не меньше длины внешнего пути дерева сравнений (суммы длин путей от корня до всех листьев), которое минимально в сбалансированном случае.

2.2.14. *Слияние упорядоченных очередей.* Разработайте статический метод, который принимает в качестве аргументов две очереди упорядоченных элементов и возвращает очередь, полученную слиянием исходных очередей в порядке возрастания.

2.2.15. *Восходящая сортировка слиянием для очередей.* Разработайте реализацию восходящей сортировки слиянием на основе следующего принципа. Из N исходных элементов создайте N очередей, каждая из одного элемента. Создайте очередь из этих N очередей. Затем многократно выполняйте операцию слияния из упражнения 2.2.14 для первых двух очередей с помещением слитой очереди в конец. Повторяйте до тех пор, пока очередь очередей не будет содержать только одну очередь.

2.2.16. *Естественная сортировка слиянием.* Напишите версию восходящей сортировки слиянием, которая использует упорядоченность массива, выполняя следующую процедуру, когда ей нужны для слияния два массива: она находит упорядоченный подмассив (увеличивая указатель, пока не встретится элемент, меньший, чем его предшественник в массиве), потом находит следующий, затем сливает их. Выразите время выполнения этого алгоритма через размер массива и количество максимальных увеличивающихся последовательностей в массиве.

2.2.17. *Сортировка связных списков.* Реализуйте естественную сортировку слиянием для связных списков. (Это самый удобный способ для сортировки связных списков, потому что он не требует дополнительной памяти и гарантированно выполняется за линейно-логарифмическое время.)

2.2.18. *Тасование связного списка.* Разработайте и реализуйте алгоритм типа "разделяй и властвуй", который случайным образом тасует связный список за линейно-логарифмическое время, используя дополнительную память логарифмического объема.

2.2.19. *Вставки.* Разработайте и реализуйте линейно-логарифмический алгоритм для вычисления количества перестановок в заданном массиве (количество обменов, необходимое сортировке вставками для этого массива — см. раздел 2.1). Эта величина связана с *тау-расстоянием Кенделла*; см. раздел 2.5.

2.2.20. *Косвенная сортировка.* Разработайте и реализуйте версию сортировки слиянием, которая не переупорядочивает массив, а возвращает массив `perm` типа `int[]` — такой, что `perm[i]` содержит индекс `i`-го наименьшего элемента в массиве.

2.2.21. *Трипликаты.* Пусть имеются три списка, содержащие каждый по N имен. Разработайте линейно-логарифмический алгоритм для определения, имеется ли имя, которое находится во всех трех списках. Если да, то нужно возвратить первое такое имя.

2.2.22. *3-частная сортировка слиянием.* Допустим, что вместо деления массива пополам на каждом шаге, он делится на трети, потом эти трети сортируются и объединяются с помощью 3-частного слияния. Каков порядок роста общего времени выполнения для этого алгоритма?

Эксперименты

2.2.23. *Усовершенствования.* Экспериментально оцените эффективность каждого из трех усовершенствований сортировки слиянием, которые описаны в тексте раздела (см. упражнение 2.2.11). Также сравните производительность реализации слияния, приведенного в тексте, со слиянием, описанным в упражнении 2.2.10. В частности, эмпирически определите наилучшее значение размера небольших подмассивов, при котором надо переключаться на сортировку вставками.

2.2.24. *Усовершенствование с помощью проверки на упорядоченность.* Экспериментально оцените для больших случайно упорядоченных массивов эффективность модификации, описанной в упражнении 2.2.8 для случайных данных. В частности, сформулируйте гипотезу о среднем количестве проверок, завершившихся успешно (т.е. массив уже упорядочен), в виде функции от N (размер исходного сортируемого массива).

2.2.25. *Многочастная сортировка слиянием.* Разработайте реализацию сортировки слиянием с помощью k-частных слияний (а не 2-частных, как обычно). Проанализируйте полученный алгоритм, сформулируйте гипотезу о лучшем значении k и экспериментально проверьте эту гипотезу.

2.2.26. *Создание массива.* Используйте программу `SortCompare` для получения грубой оценки влияния на производительность на вашем компьютере создания массива `aux[]` в методе `merge()`, а не в `sort()`.

2.2.27. *Длины подмассивов.* С помощью выполнения сортировки слиянием для больших случайных массивов эмпирически определите среднюю длину другого подмассива, когда в первом подмассиве заканчиваются данные, в виде функции от N (суммы размеров двух подмассивов для данного слияния).

2.2.28. *Нисходящие и восходящие слияния.* Используйте программу `SortCompare` для сравнения нисходящей и восходящей сортировки слиянием для $N = 10^3$, 10^4, 10^5 и 10^6.

2.2.29. *Естественная сортировка слиянием.* Эмпирически определите количество проходов, необходимое для естественной сортировки слиянием (см. упражнение 2.2.16) для случайных ключей `Long` и $N = 10^3$, 10^6 и 10^9. *Подсказка*: для выполнения этого упражнения не обязательно реализовывать сортировку (и даже генерировать полные 64-битовые ключи).

2.3. Быстрая сортировка

В настоящем разделе мы рассмотрим алгоритм сортировки, который, пожалуй, используется чаще других — это *быстрая сортировка*. Ее популярность объясняется тем, что она несложна для реализации, хорошо работает для множества различных видов входных данных, и в обычных ситуациях значительно быстрее любых других методов сортировки. Сильные стороны быстрой сортировки: она выполняется на месте (использует только небольшой вспомогательный стек) и упорядочивает массив длиной N в среднем за время, пропорциональное $N\lg N$. Ни один из рассмотренных нами до сих пор алгоритмов не обладает таким набором свойств. Кроме того, у быстрой сортировки более короткий внутренний цикл по сравнению с большинством других алгоритмов сортировки, а это означает, что она быстро работает не только в теории, но и на практике. Ее основной недостаток — излишняя чувствительность: при реализации быстрой сортировки необходима определенная осторожность, чтобы не нанести ущерба производительности. В литературе описаны многочисленные примеры ошибок, которые приводили на практике к квадратичной производительности. К счастью, из этих ошибок извлечены уроки, оформленные в различные усовершенствования алгоритма — как мы увидим, это еще больше увеличивает его полезность.

Базовый алгоритм

Быстрая сортировка использует для упорядочения элементов метод "разделяй и властвуй". Она работает, выполняя *разбиения* массива на два подмассива, с последующей независимой сортировкой полученных подмассивов. В этом смысле быстрая сортировка является дополнением сортировки слиянием: при сортировке слиянием мы разбиваем массив на два подмассива, которые затем упорядочиваются и объединяются с получением полного упорядоченного массива; а при быстрой сортировке мы переупорядочиваем массив так, что после сортировки двух полученных подмассивов исходный массив также становится упорядоченным. В первом случае два рекурсивных вызова выполняются *до* обработки всего массива, а во втором два рекурсивных вызова осуществляются *после* обработки всего массива. При сортировке слиянием массив делится пополам, а при быстрой сортировке позиция разбиения зависит от содержимого массива (рис. 2.3.1).

Входные данные	Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
Тасование	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
Разбиение	E	C	A	I	E	**K**	L	P	U	T	M	Q	R	X	O	S
Сортировка левой части	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
Сортировка правой части	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
Результат	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

центральный элемент

не больше   *не меньше*

Рис. 2.3.1. Принцип работы быстрой сортировки

Основа метода — процесс разбиения, который переупорядочивает массив так, чтобы выполнялись три следующих условия:

- элемент a[j] находится в массиве на своем окончательном месте (для некоторого j);
- ни один элемент от a[lo] до a[j-1] не больше, чем a[j];
- ни один элемент от a[j+1] до a[hi] не меньше, чем a[j].

Для полного упорядочения всего массива выполняется разбиение, а затем рекурсивное применение метода.

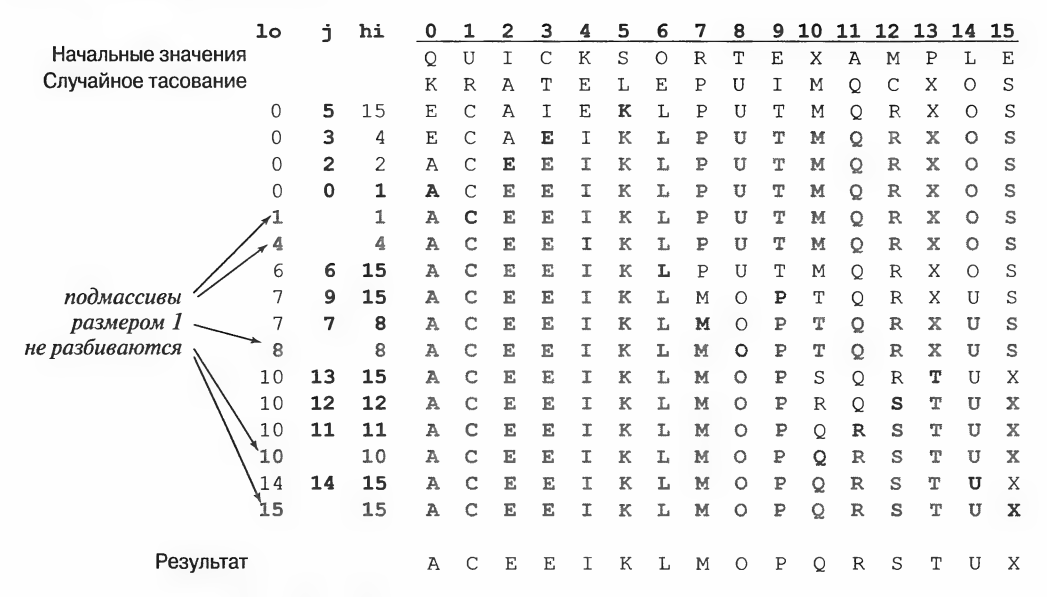
Поскольку процесс разбиения всегда помещает один элемент на свое место в массиве, нетрудно разработать формальное доказательство по индукции, что рекурсивный метод выполняет правильное упорядочение. Если левый и правый подмассивы правильно упорядочены, то упорядочен будет и результирующий массив, который состоит из левого подмассива (упорядочен и ни один элемент не больше центрального элемента), центрального элемента (по которому выполнено разбиение) и правого подмассива (упорядочен и ни один элемент не меньше центрального элемента). Алгоритм 2.5 (см. листинг 2.3.1 и рис. 2.3.2) является рекурсивной программой, в которой реализована описанная идея. Это *рандомизированный* алгоритм, т.к. он перед началом сортировки перемешивает массив случайным образом. Мы делаем это для того, чтобы можно было предсказать характеристики производительности (а потом и воспользоваться ими).

Листинг 2.3.1. Алгоритм 2.5. Быстрая сортировка

```
public class Quick
{
  public static void sort(Comparable[] a)
  {
    StdRandom.shuffle(a);        // Устранение зависимости от входных данных.
    sort(a, 0, a.length - 1);
  }
  private static void sort(Comparable[] a, int lo, int hi)
  {
    if (hi <= lo) return;
    int j = partition(a, lo, hi);     // Разбиение (см. листинг 2.3.2).
    sort(a, lo, j-1);                 // Сортировка левой части a[lo .. j-1].
    sort(a, j+1, hi);                 // Сортировка правой части a[j+1 .. hi].
  }
}
```

Быстрая сортировка рекурсивно упорядочивает подмассив a[lo...hi] с помощью метода partition(), который помещает a[i] на свое место и распределяет остальные элементы так, чтобы рекурсивные вызовы завершили сортировку.

Чтобы получить завершенную программу, необходимо еще реализовать метод разбиения. Мы будем использовать следующую общую стратегию. Сначала произвольно выбираем в качестве *центрального элемента* значение a[lo] — оно попадет в свою окончательную позицию. Затем просматриваем элементы массива с левого конца, пока не найдем элемент, который больше (или равен) центрального элемента, и просматриваем элементы массива с правого конца массива, пока не найдем элемент, который меньше (или равен) центрального элемента (рис. 2.3.3).



	lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Начальные значения				Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
Случайное тасование				K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
	0	5	15	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
	0	3	4	E	C	A	E	I	K	L	P	U	T	M	Q	R	X	O	S
	0	2	2	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	0	0	1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	1		1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	4		4	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	6	6	15	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	7	9	15	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	7	7	8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	8		8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	10	13	15	A	C	E	E	I	K	L	M	O	P	S	Q	R	T	U	X
	10	12	12	A	C	E	E	I	K	L	M	O	P	R	Q	S	T	U	X
	10	11	11	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	10		10	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	14	14	15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	15		15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
Результат				A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

Рис. 2.3.2. Трассировка выполнения быстрой сортировки

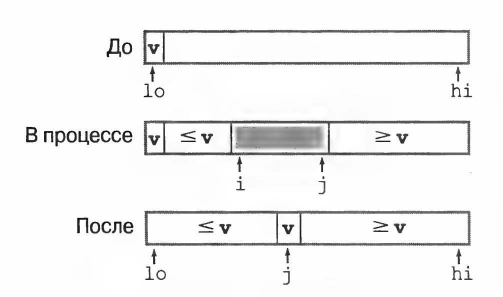


Рис. 2.3.3. Принцип разбиения в быстрой сортировке

Элементы, на которых остановился просмотр, находятся не на своем месте в окончательном разбитом массиве, поэтому мы обмениваем их местами. Продолжая эти действия, мы обеспечиваем, что ни один элемент слева от левого индекса i не больше центрального элемента, и ни один элемент справа от правого индекса j не меньше центрального элемента. Когда индексы просмотра встретятся, остается только завершить процесс разбиения и обменять центральный элемент a[lo] с самым правым элементом из левого подмассива (a[j]) и возвратить его индекс j.

С реализацией быстрой сортировки связан ряд тонких моментов, которые отражены в нашем коде и достойны упоминания, т.к. каждый из них может привести к неверному коду или существенно ухудшить производительность. Сейчас мы рассмотрим некоторые из этих моментов, а позже в данном разделе обсудим три важных высокоуровневых усовершенствования.

Листинг 2.3.2. Разбиение для быстрой сортировки

```
private static int partition(Comparable[] a, int lo, int hi)
{  // Разбиение на a[lo..i-1], a[i], a[i+1..hi].
   int i = lo, j = hi+1;             // Левый и правый индексы просмотра
   Comparable v = a[lo];             // Центральный элемент
   while (true)
   {  // Просмотр справа, просмотр слева, проверка на завершение и обмен.
      while (less(a[++i], v)) if (i == hi) break;
      while (less(v, a[--j])) if (j == lo) break;
```

```
      if (i >= j) break;
      exch(a, i, j);
   }
   exch(a, lo, j);        // Помещение v = a[j] на свое место
   return j;              // так, что a[lo..j-1] <= a[j] <= a[j+1..hi].
}
```

Этот код разбивает массив a[] по значению v из a[lo]. Основной цикл завершается при встрече индексов i и j. В цикле i увеличивается, пока a[i] меньше v, a j уменьшается, пока a[j] больше v, и затем выполняется обмен элементов, чтобы сохранить инвариант, что слева от i нет элементов больше v, а справа от j нет элементов меньше v. После встречи индексов разбиение завершается, для чего осуществляется обмен a[lo] с a[j] (и, значит, центральное значение заносится в a[j] — рис. 2.3.4).

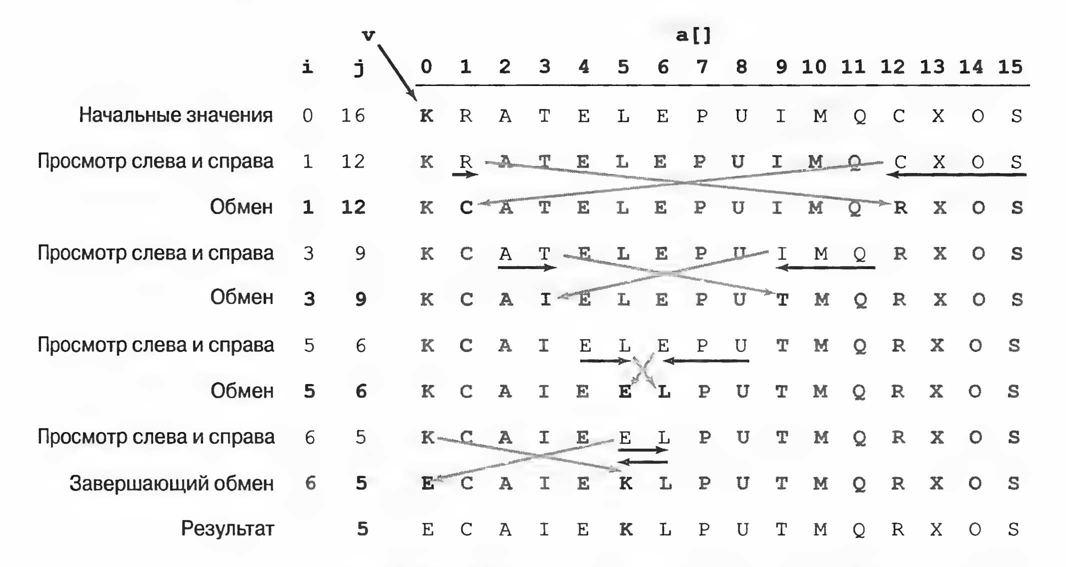


Рис. 2.3.4. Трассировка разбиений (содержимое массива до и после каждого обмена)

Разбиение на месте

При использовании дополнительного массива разбиение реализовать легко — но не настолько легко, что это перекрывает дополнительные затраты на копирование разбитой версии массива на исходное место. Неопытный Java-программист может даже создавать новый массив в рекурсивном методе для каждого разбиения, что значительно замедлит сортировку.

Отслеживание границ

Если в качестве центрального элемента берется наименьший или наибольший элемент массива, то приходится следить, чтобы указатели не вышли за левый или правый конец массива соответственно. В нашей реализации partition() присутствуют явные проверки для защиты от такой возможности. Проверка (j==lo) излишня, т.к. центральный элемент находится в a[lo] и не может быть меньше себя. Аналогичный прием для правого конца позволяет избавиться от обеих проверок (см. упражнение 2.3.17).

Сохранение случайности

Тасование перемешивает массив случайным образом. Поскольку оно одинаково обрабатывает все элементы в подмассивах, алгоритм 2.5 обладает свойством, что его два подмассива также случайно упорядочены. Этот факт важен для предсказания времени выполнения алгоритма. Другой способ сохранить случайность — выбор случайного элемента для разбиения в методе `partition()`.

Завершение цикла

Опытные программисты всегда внимательно относятся к критерию завершения любого цикла, и цикл разбиения для быстрой сортировки не является исключением. Правильная проверка, встретились ли указатели просмотра, не так проста, как может сначала показаться. Программисты часто забывают, что массив может содержать и другие элементы с тем же значением ключа, что и центральный элемент.

Обработка элементов с ключами, равными ключу центрального элемента

Лучше останавливать просмотр слева при встрече элементов с ключом, который больше *или равен* ключу центрального элемента, и останавливать просмотр справа при встрече с ключом, который меньше *или равен* ключу центрального элемента — как это сделано в алгоритме 2.5. Это правило может привести к ненужным обменам элементов с ключами, равными ключу центрального элемента, но оно необходимо, чтобы избежать квадратичного времени выполнения в некоторых типичных приложениях (см. упражнение 2.3.11). Ниже будет рассмотрена лучшая стратегия для случая, когда массив содержит большое количество элементов с одинаковыми ключами.

Завершение рекурсии

Опытные программисты также внимательно следят, чтобы любой рекурсивный метод обязательно завершался — и быстрая сортировка также не является исключением. Например, часто встречается ошибка, когда центральный элемент не перемещается на свое место; тогда если центральным элементом выбирается наибольший или наименьший элемент массива, программа уходит в бесконечный рекурсивный цикл.

Характеристики производительности

Быстрая сортировка подверглась очень тщательному математическому анализу, потому известно много точных сведений о ее производительности. Этот анализ проверен обширными эмпирическими сведениями и полезен для настройки алгоритма на оптимальную производительность.

Внутренний цикл в быстрой сортировке (в методе разбиения) увеличивает индекс и сравнивает элемент массива с фиксированным значением. Эта простота является одной из причин, которая делает быструю сортировку действительно быстрой: трудно придумать более короткий внутренний цикл в алгоритме сортировки. Например, сортировка слиянием и сортировка Шелла обычно работают медленнее быстрой сортировки, т.к. они выполняют перемещения данных в своих внутренних циклах.

Вторая причина высокой скорости быстрой сортировки в том, что она использует немного сравнений. В конце концов, эффективность сортировки зависит от того, как разбиение делит массив на части, а это зависит от значения ключа центрального элемента. Разбиение делит большой случайно упорядоченный массив на два меньших случайно упорядоченных подмассива, но реальная точка разбиения может с одинаковой вероят-

ностью (для различных ключей) оказаться в любом месте массива. Ниже мы проанализируем, чем этот выбор отличается от идеального выбора.

Лучшим для быстрой сортировки является случай, когда на каждом этапе разбиения массив делится точно пополам. При этом количество сравнений, выполняемых быстрой сортировкой, удовлетворяет рекуррентному соотношению, характерному для принципа “разделяй и властвуй”: $C_N = 2C_{N/2} + N$. Слагаемое $2C_{N/2}$ означает стоимость сортировки двух подмассивов, а N — стоимость просмотра каждого элемента с помощью левого или правого индекса разбиения. В доказательстве утверждения Е для сортировки слиянием уже было показано, что это рекуррентное уравнение имеет решение $C_N \sim N\lg N$. Реальные условия не всегда так благоприятны, но все же разбиение попадает в середину массива *в среднем*. Учет точной вероятности каждой позиции разбиения усложняет как само рекуррентное выражение, так и его решение, но окончательный результат остается похожим. Доказательство этого результата является основанием нашей веры в быструю сортировку. Если у вас нет особой любви к математическим вычислениям, вы можете просто пропустить их (приняв на веру), а при наличии такой любви доказательство может показаться интересным.

Утверждение Л. Для упорядочения массива длиной N с различными ключами быстрая сортировка выполняет $\sim 2N\lg N$ сравнений (и в шесть раз меньше обменов).

Доказательство. Пусть C_N — среднее количество сравнений, которое необходимо для упорядочения N элементов с различными значениями. В крайних случаях $C_0 = C_1 = 0$, а для $N > 1$ можно записать рекуррентное соотношение, которое непосредственно соответствует рекурсивной программе:

$$C_N = N + 1 + (C_0 + C_1 + \ldots + C_{N-2} + C_{N-1}) \,/\, N + (C_{N-1} + C_{N-2} + \ldots + C_0) \,/\, N$$

Первое слагаемое означает стоимость разбиения (всегда $N + 1$), второе — среднюю стоимость упорядочения левого подмассива (который с одинаковой вероятностью может иметь любой размер от 0 до $N-1$), а третье — среднюю стоимость упорядочения правого подмассива (равна стоимости для левого подмассива). Умножение на N и приведение подобных членов дает

$$NC_N = N(N + 1) + 2(C_0 + C_1 + \ldots + C_{N-2} + C_{N-1})$$

Вычтя это выражение из такого же равенства для $N-1$, получим

$$NC_N - (N-1)C_{N-1} = 2N + 2C_{N-1}$$

Перегруппировка членов и деление на $N(N + 1)$ дает равенство

$$C_N/(N + 1) = 2C_{N-1}/N + 2/(N + 1)$$

которое развертывается в такой результат:

$$C_N \sim 2(N + 1)(1/3 + 1/4 + \ldots + 1/(N + 1))$$

Величина в скобках представляет собой дискретную оценку площади под кривой $2/x$ от 3 до $N+1$, и интегрирование дает $C_N \sim 2N\ln N$. Поскольку $2N\ln N \approx 1{,}39N\lg N$, то среднее количество сравнений лишь на 39% выше, чем в наилучшем случае.

Аналогичный (но более сложный анализ) дает количество обменов.

Если ключи не обязательно различны, как в типичных реальных ситуациях, точный анализ существенно усложняется, но нетрудно показать, что среднее количество сравнений не превышает C_N даже при наличии повторяющихся ключей (немного ниже мы рассмотрим способ *усовершенствования* быстрой сортировки в этом случае).

Несмотря на многочисленные достоинства, базовой программе быстрой сортировки присущ один потенциальный недостаток: она может крайне неэффективно работать, если разбиения не будут сбалансированы. Например, может случиться так, что для первого разбиения будет взят наименьший элемент, для второго разбиения — второй наименьший элемент и т.д. Тогда при каждом вызове программа будет удалять лишь один элемент, что приведет к обилию разбиений больших подмассивов. Именно для этого перед быстрой сортировкой мы выполняем случайное тасование массива. Такое действие делает последовательное появление неудачных разбиений настолько невероятным, что о нем можно не беспокоиться.

Утверждение М. Быстрая сортировка выполняет в худшем случае $\sim N^2/2$ сравнений, но случайное тасование защищает от такой возможности.

Доказательство. Согласно приведенному выше рассуждению, количество сравнений, если в каждом разбиении один из подмассивов пуст, равно

$$N + (N + 1) + (N + 2) + \ldots + 2 + 1 = (N + 1)N / 2$$

Это означает не только, что время выполнения квадратично, но и что для обработки рекурсии понадобится линейный объем памяти — а это неприемлемо для больших массивов. Однако немного более подробный анализ показывает, что среднеквадратичное отклонение количества сравнений имеет порядок $0{,}65N$, поэтому при увеличении N время выполнения стремится к среднему значению и вряд ли намного отклонится от него. К примеру, даже грубая оценка на основе неравенства Чебышева показывает, что вероятность того, что время сортировки миллиона элементов в десять раз превысит среднее время, меньше 0,00001 (а реальная вероятность значительно меньше). Вероятность того, что время выполнения для большого массива будет квадратичным, настолько мизерна, что такой вариант можно спокойно игнорировать (см. упражнение 2.3.10). Например, вероятность того, что при упорядочении большого массива быстрая сортировка выполнит на вашем компьютере столько же сравнений, сколько сортировка вставками или выбором, гораздо меньше, чем вероятность, что во время сортировки компьютер расплавится от удара молнии!

В общем, можно быть уверенным, что время сортировки алгоритмом 2,5 N элементов будет отличаться от $1{,}39N\lg N$ лишь на постоянный множитель. Это справедливо и для сортировки слиянием, но быстрая сортировка обычно работает быстрее, поскольку она выполняет значительно меньше перемещений данных (хотя и на 39% больше сравнений). Это вероятностное обоснование, но на него вполне можно положиться.

Алгоритмические усовершенствования

Быстрая сортировка была изобретена в 1960 г. Хоаром (C. A. R. Hoare), и с тех пор очень многие люди изучали и улучшали ее. Всегда интересно попытаться еще более ускорить быструю сортировку: в вычислительной технике не переводятся желающие найти более быстрый сортирующий алгоритм, и особенно в отношении быстрой сортировки.

Почти сразу после опубликования Хоаром первоначального варианта алгоритма исследователи начали предлагать различные способы его усовершенствования. Не все из них оказались удачными, т.к. алгоритм настолько удачно сбалансирован, что многие улучшения приводят к непредвиденным побочным эффектам. Однако некоторые из них оказались довольно эффективными, и мы сейчас их рассмотрим.

Если ваш код сортировки предполагается использовать много раз либо для сортировки очень больших массивов (или, особенно, если его предполагается сделать библиотечной сортировкой, которая будет применяться для упорядочения массивов с заранее не известными характеристиками), то стоит рассмотреть усовершенствования, которые будут описаны в нескольких следующих параграфах. Как уже было сказано, следует экспериментально определить эффективность этих усовершенствований и наилучший набор параметров для вашей реализации. Обычно эффективность повышается на 20–30%.

Отсечка на сортировку вставками

Как и в случае большинства рекурсивных алгоритмов, легкий способ повысить производительность быстрой сортировки основан на следующих двух наблюдениях.

- Для маленьких подмассивов быстрая сортировка работает медленнее, чем сортировка вставками.
- В силу рекурсивности метод `sort()` быстрой сортировки непременно вызывает себя для маленьких подмассивов.

Поэтому имеет смысл для маленьких подмассивов переключаться на сортировку вставками. Это нетрудно сделать в алгоритме 2.5: достаточно заменить в методе `sort()` оператор

```
if (hi <= lo) return;
```

оператором, который вызывает сортировку вставками для небольших подмассивов:

```
if (hi <= lo + M) {  Insertion.sort(a, lo, hi); return;  }
```

Оптимальное значение отсечки `M` зависит от системы, но в большинстве случаев должно хорошо работать любое значение от 5 до 15 (см. упражнение 2.3.25).

Разбиение по медиане из трех

Второй легкий способ повысить производительность быстрой сортировки предусматривает использование в качестве центрального элемента медианы из небольшой выборки элементов, взятых из подмассива (рис. 2.3.5). Это несколько улучшает качество разбиений, но за счет вычисления медианы. Оказывается, удобнее использовать выборку из трех элементов, а затем выполнять разбиение по среднему элементу (см. упражнения 2.3.18 и 2.3.19). В качестве бонуса элементы выборки можно использовать в качестве сигнальных маркеров на концах массива — это позволяет удалить в методе `partition()` обе проверки на выход за границы массива.

Сортировка с оптимальной энтропией

Массивы с большим количеством повторяющихся ключей часто встречаются в приложениях. Например, может понадобиться отсортировать большой файл персонала по годам рождения или, возможно, разделить мужчин и женщин. В таких ситуациях предложенную нами реализацию быстрой сортировки можно существенно улучшить.

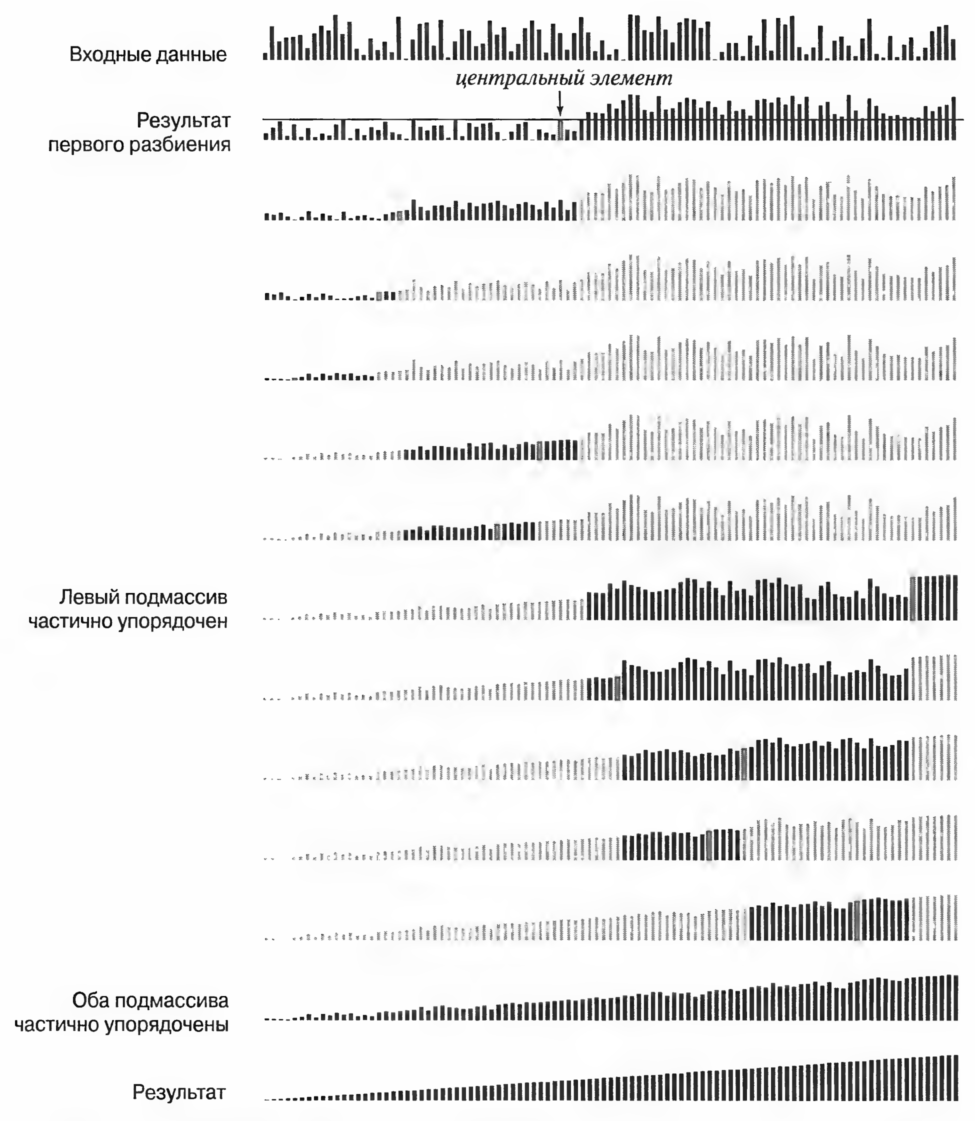


Рис. 2.3.5. *Быстрая сортировка с разбиением по медиане из трех и отсечкой для небольших подмассивов*

Например, подмассив, который состоит только из одинаковых элементов (одно и то же значение ключа), можно уже не обрабатывать, но наша реализация продолжит разбиения до самых маленьких подмассивов. В ситуациях, когда входной массив содержит большое количество одинаковых ключей, из-за рекурсивной природы быстрой сортировки будут часто встречаться подмассивы, состоящие только из элементов с одинаковыми ключами. Это дает возможность для серьезного усовершенствования — от линейно-логарифмической производительности уже известных нам реализаций до линейной.

В рассматриваемом случае удобно разбивать массив на *три* части: с ключами, меньшими, равными и большими ключа центрального элемента (рис. 2.3.6 и 2.3.7).

lt	i	gt	0	1	2	3	4	5	6	7	8	9	10	11
0	0	11	R	B	W	W	R	W	B	R	R	W	B	R
0	1	11	R	B	W	W	R	W	B	R	R	W	B	R
1	2	11	B	R	W	W	R	W	B	R	R	W	B	R
1	2	10	B	R	R	W	R	W	B	R	R	W	B	W
1	3	10	B	R	R	W	R	W	B	R	R	W	B	W
1	3	9	B	R	R	B	R	W	B	R	R	W	W	W
2	4	9	B	B	R	R	R	W	B	R	R	W	W	W
2	5	9	B	B	R	R	R	W	B	R	R	W	W	W
2	5	8	B	B	R	R	R	W	B	R	R	W	W	W
2	5	7	B	B	R	R	R	R	B	R	W	W	W	W
2	6	7	B	B	R	R	R	R	B	R	W	W	W	W
3	7	7	B	B	B	R	R	R	R	R	W	W	W	W
3	8	7	B	B	B	R	R	R	R	R	W	W	W	W
3	8	7	B	B	B	R	R	R	R	R	W	W	W	W

Рис. 2.3.6. Трассировка трехчастного разбиения (содержимое массива после каждого выполнения цикла)

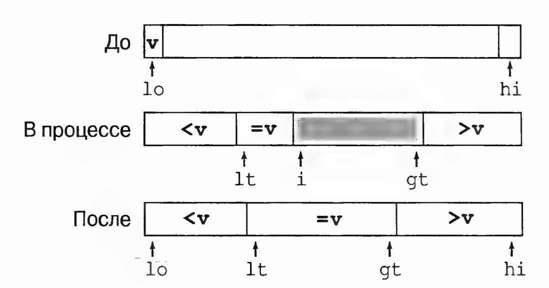


Рис. 2.3.7. Принцип трехчастного разбиения

Выполнение такого разбиения сложнее, чем описанное ранее 2-частное разбиение, и для него предложены различные методы. Это было классическое упражнение по программированию, известное благодаря Э. Дейкстре (E. W. Dijkstra) как задача *датского национального флага* — потому что она похожа на сортировку массива с тремя возможными значениями ключей, которые могут соответствовать трем цветам флага.

Решение Дейкстры для этой задачи приводит к невероятно простому коду разбиения, приведенному в листинге 2.3.3. Оно основано на простом просмотре массива слева направо, которое использует указатель `lt` такой, что `a[lo..lt-1]` *меньше* `v`, указатель `gt` такой, что `a[gt+1,hi]` *больше* `v`, и указатель `i` такой, что `a[lt..i-1]` *равны* `v`, а элементы `a[i..gt]` еще не просмотрены. Сначала `i` равно `lo`, и для каждого `a[i]` выполняется трехчастное сравнение, предусмотренное в интерфейсе `Comparable` (вместо метода `less()`) — т.е. непосредственно обрабатываются три возможных случая:

- `a[i]` меньше `v`: выполняется обмен `a[lt]` с `a[i]` и увеличиваются указатели `lt` и `i`;
- `a[i]` больше `v`: выполняется обмен `a[i]` с `a[gt]` и уменьшается `gt`;
- `a[i]` равно `v`: увеличивается `i`.

Листинг 2.3.3. Быстрая сортировка с трехчастным разбиением

```
public class Quick3way
{
   private static void sort(Comparable[] a, int lo, int hi)
   {  // Общедоступный метод sort(), вызывающий этот метод, см. в листинге 2.3.1.
      if (hi <= lo) return;
      int lt = lo, i = lo+1, gt = hi;
      Comparable v = a[lo];
      while (i <= gt)
      {
         int cmp = a[i].compareTo(v);
         if      (cmp < 0) exch(a, lt++, i++);
         else if (cmp > 0) exch(a, i, gt--);
         else              i++;
      }  // Теперь a[lo..lt-1] < v = a[lt..gt] < a[gt+1..hi].
      sort(a, lo, lt - 1);
      sort(a, gt + 1, hi);
   }
}
```

Этот код выполняет разбиение, помещая на свое место ключи, равные центральному элементу; поэтому он не включает эти ключи в подмассивы для рекурсивных вызовов. Данная реализация работает гораздо эффективнее стандартной реализации быстрой сортировки для массивов с большим количеством повторяющихся ключей (см. текст).

Каждая из этих операций поддерживает инвариант и уменьшает значение `gt-i` (чтобы завершить цикл). Кроме того, каждый просматриваемый элемент приводит к обмену — *кроме* элементов с ключами, равными ключу центрального элемента.

Этот код был разработан вскоре после опубликования быстрой сортировки в 1970-х годах, но не пользовался популярностью, т.к. он выполнял значительно больше обменов, чем стандартный 2-частный метод разбиения для общего случая, когда количество повторяющихся ключей в массиве не очень велико.

В 1990-х годах Дж. Бентли (J. Bentley) и Д. Мак-Илрой (D. McIlroy) предложили хитроумную реализацию, которая устраняет эту проблему (см. упражнение 2.3.22), и заметили, что в практических ситуациях с большим количеством равных ключей трехчастное разбиение делает быструю сортировку асимптотически более быстрой, чем сортировка слиянием и другие методы. Позже Дж. Бентли и Р. Седжвик разработали доказательство этого факта, которое мы сейчас рассмотрим.

Но ведь мы доказали, что сортировка слиянием оптимальна. Как же получилось прорваться сквозь нижнюю границу? Ответ на этот вопрос состоит в том, что утверждение И в разделе 2.2 имеет дело с производительностью в случае худших данных из всех возможных, а сейчас мы рассматриваем производительность в худшем случае при наличии некоторой информации об обрабатываемых ключах. Сортировка слиянием не гарантирует оптимальной производительности для любого заданного распределения повторяющихся ключей во входных данных — например, сортировка слиянием имеет линейно-логарифмическую сложность для случайно упорядоченного массива, содержащего лишь постоянное количество различных значений ключей, а быстрая сортировка с трехчастным разбиением выполняется для такого массива за линейное время. И действительно, визуальная трассировка на рис. 2.3.8 показывает, что для времени выполнения осторожной оценкой является N-кратное количество значений ключей.

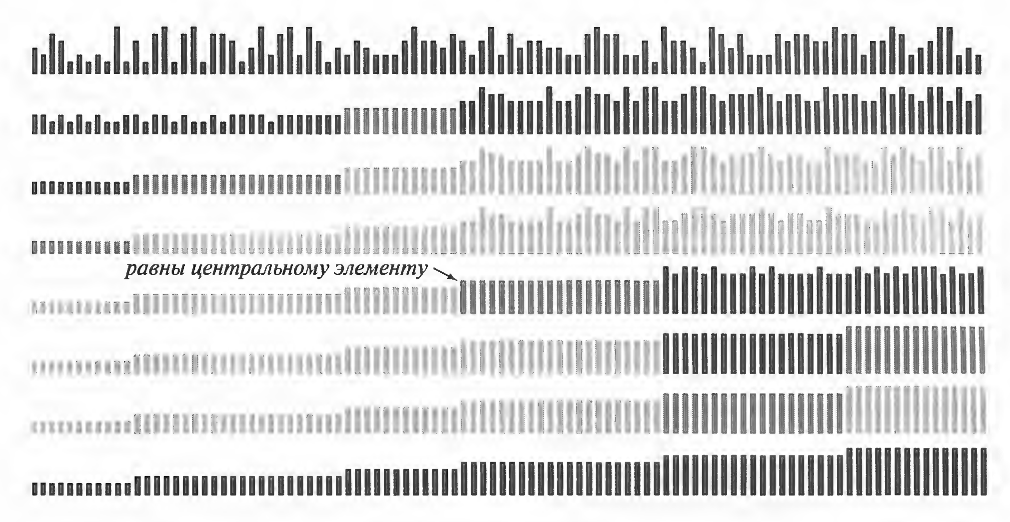


Рис. 2.3.8. Визуальная трассировка быстрой сортировки с трехчастным разбиением

Анализ, который облекает эти рассуждения в точные формулы, учитывает распределение значений ключей. Пусть имеются N ключей с k различными значениями ключей. Введем обозначения: f_i — частота появления i-го значения ключа, а $p_i = f_i/N$ — вероятность появления i-го значения ключа при выборке случайного элемента массива. *Энтропия Шеннона* для ключей (классическая мера содержащейся в них информации) определяется так:

$$H = -(p_1 \lg p_1 + p_2 \lg p_2 + \ldots + p_k \lg p_k)$$

Если массив элементов упорядочен, можно вычислить его энтропию, подсчитав частоту появления каждого значения ключа. Интересно, что из этой энтропии можно также получить нижнюю и верхнюю границы для количества сравнений, выполняемых быстрой сортировкой с трехчастным разбиением.

Утверждение Н. Ни один алгоритм сортировки на основе сравнений не может гарантированно упорядочить N элементов с помощью менее чем $NH-N$ сравнений, где H — энтропия Шеннона, вычисленная по частотам значений ключей.

Набросок доказательства. Этот результат следует из (относительно нетрудного) обобщения доказательства для нижней границы из утверждения И в разделе 2.2.

Утверждение О. Для упорядочения N элементов быстрая сортировка с трехчастным разбиением выполняет $\sim(2\ln 2)NH$ сравнений, где H — энтропия Шеннона, вычисленная по частотам значений ключей.

Набросок доказательства. Этот результат следует из (относительно сложного) обобщения анализа среднего случая для быстрой сортировки из утверждения Л. Как и в случае различных ключей, эта стоимость примерно на 39% больше оптимальной (но не более чем на постоянный коэффициент).

Если все ключи различны (все вероятности равны $1/N$), то $H = \lg N$, что согласуется с утверждением И в разделе 2.2 и утверждением Л. Худший случай для трехчастного разбиения — когда все ключи различны; а при наличии повторяющихся ключей оно рабо-

тает значительно лучше, чем сортировка слиянием. Более важно то, что эти два свойства вместе означают, что быстрая сортировка с трехчастным разбиением имеет *оптимальную энтропию* — т.е. для любого конкретного распределения значений входных ключей среднее количество сравнений, выполняемых наилучшим алгоритмом сортировки на основе сравнений, и среднее количество сравнений, выполняемых трехчастной быстрой сортировкой, отличаются только на постоянный множитель.

Как и в случае стандартной быстрой сортировки, при увеличении размера массива время выполнения стремится к среднему значению, и значительные отклонения от среднего крайне маловероятны. Поэтому вполне можно считать, что время выполнения трехчастной быстрой сортировки пропорционально произведению N и энтропии распределения значений входных ключей. Это свойство алгоритма практически важно, т.к. ***для массивов с большим количеством повторяющихся ключей время выполнения сортировки сводится от линейно-логарифмического к линейному.*** Порядок расположения ключей не важен, поскольку алгоритм тасует их для защиты от худшего случая. Энтропия определяется распределением ключей, и ни один алгоритм на основе сравнений не может использовать меньше сравнений, чем это определено энтропией. Эта возможность применения повторов во входных данных делает трехчастную быструю сортировку оптимальным выбором для библиотечной сортировки: клиенты, сортирующие массивы с большим количеством повторяющихся ключей, встречаются довольно часто.

Тщательно настроенная версия быстрой сортировки обычно работает значительно быстрее любого другого метода сортировки на основе сравнений — на большинстве компьютеров и для большинства приложений. В современной вычислительной инфраструктуре быстрая сортировка применяется очень широко, т.к. рассмотренные нами математические модели подсказывают, что она обгоняет другие методы в реальных случаях, а обширные эксперименты и практические наблюдения за несколько десятков лет подтверждают этот вывод.

В главе 5 мы увидим, что разработка алгоритмов еще не остановилась на этом, и можно изобрести алгоритмы, в которых сравнения вообще не используются! Но, оказывается, и в этом случае лучше всех вариант быстрой сортировки!

Вопросы и ответы

Вопрос. **А нельзя ли как-то просто делить массив пополам, а не брать центральный элемент, который может попасть куда угодно?**

Ответ. Этот вопрос не давал покоя экспертам больше десяти лет. Поиск *медианного* значения ключа в массиве с последующим разбиением по этому значению слишком трудоемок. Задача поиска медианы будет рассмотрена в разделе 2.5. Медиану можно найти за линейное время, но стоимость известных алгоритмов (которые основаны на разбиении для быстрой сортировки) значительно превышает 39% экономии от деления массива на равные части.

Вопрос. **Случайное тасование массива должно существенно увеличивать общее время сортировки. Действительно ли стоит его выполнять?**

Ответ. Да. Оно защищает от худшего случая и делает время выполнения предсказуемым. Этот подход предложил еще Хоар, когда опубликовал свой алгоритм в 1960 г. — характерный (и один из первых) рандомизированный алгоритм.

Вопрос. **Зачем уделять так много внимания равным ключам?**

Ответ. Этот вопрос непосредственно влияет на производительность в реальных ситуациях. На него почему-то не обращали внимания десятки лет, и в результате некоторые старые реализации быстрой сортировки имели квадратичное время выполнения для массивов с большим количеством равных ключей, которые довольно часто встречаются в практических приложениях. Более удачные реализации, подобные алгоритму 2.5, упорядочивают такие массивы за линейно-логарифмическое время, но улучшение его до линейного, как в случае сортировки с оптимальной энтропией в конце данного раздела, может пригодиться во многих ситуациях.

Упражнения

2.3.1. Покажите, в стиле трассировки для метода `partition()`, как этот метод разбивает массив `E A S Y Q U E S T I O N`.

2.3.2. Покажите, в стиле трассировки, приведенной в этом разделе для быстрой сортировки, как быстрая сортировка упорядочивает массив `E A S Y Q U E S T I O N` (не выполняйте в этом упражнении предварительное перемешивание).

2.3.3. Каково максимальное количество обменов с наибольшим элементом, которое может быть выполнено при упорядочивании методом `Quick.sort()` массива длиной N?

2.3.4. Предположим, что предварительное перемешивание не выполняется. Приведите шесть массивов из десяти элементов, для которых метод `Quick.sort()` выполняет максимальное количество сравнений.

2.3.5. Приведите фрагмент кода, который упорядочивает массив, если в нем содержатся лишь два различных значения ключей.

2.3.6. Напишите программу для вычисления точного значения C_N и сравните его с аппроксимацией $2N\ln N$ для $N = 100$, 1000 и 10000.

2.3.7. Найдите ожидаемое количество подмассивов размером 0, 1 и 3 при упорядочении быстрой сортировкой массива, содержащего N различных ключей. При наличии математической подготовки найдите его теоретически; иначе проведите эксперименты и предложите гипотезу.

2.3.8. Сколько примерно сравнений понадобится методу `Quick.sort()` для упорядочения массива, содержащего N одинаковых ключей?

2.3.9. Что произойдет, если на вход `Quick.sort()` подать массив, содержащий элементы лишь с двумя различными ключами? А с тремя различными ключами?

2.3.10. Согласно *неравенству Чебышева*, вероятность, что случайная переменная отклоняется от среднего значения более чем на k среднеквадратичных отклонений, меньше $1/k^2$. Найдите с помощью этого неравенства границу вероятности, что количество сравнений, выполненных быстрой сортировкой для упорядочения $N=1$ миллиона элементов, превышает 100 миллиардов $(0{,}1N^2)$.

2.3.11. Пусть при встрече ключей, равных ключу центрального элемента, просмотры не останавливаются, а пропускают эти элементы. Покажите, что для всех массивов с лишь постоянным количеством различных ключей время выполнения такого варианта быстрой сортировки квадратично.

2.3.12. Покажите, в стиле трассировки, приведенной в этом разделе, как сортировка с оптимальной энтропией выполняет первое разбиение массива `B A B A B A B A C A D A B R A`.

2.3.13. *Рекурсивная глубина* быстрой сортировки в лучшем, худшем и среднем случаях — это размер стека, необходимого системе для обслуживания рекурсивных вызовов. С помощью упражнения 2.3.20 предложите способ гарантировать логарифмическую рекурсивную глубину в худшем случае.

2.3.14. Докажите, что при упорядочении быстрой сортировкой массива с N различными элементами вероятность сравнения i-го и j-го наибольших элементов равна $2/(j-i)$. Используйте этот результат для доказательства утверждения Л.

Творческие задачи

2.3.15. *Болты и гайки* (G. J. E. Rawlins). Пусть имеется куча перемешанных N болтов и N гаек, и нужно быстро найти соответствующие пары болтов и гаек. Для каждого болта есть точно одна гайка, и для каждой гайки есть точно один болт. Попробовав навинтить гайку на болт, можно увидеть, что из них больше, но невозможно непосредственно сравнить два болта или две гайки. Приведите эффективный метод решения этой задачи.

2.3.16. *Лучший случай.* Напишите программу, которая генерирует наилучший массив (без повторов) для метода `sort()` в алгоритме 2.5 — массив из N элементов с различными ключами, такой, что в результате каждого разбиения получаются подмассивы, размеры которых отличаются не более чем на 1 (такие же размеры подмассивов получаются и для массива с N одинаковыми ключами). В этом упражнении не выполняйте предварительное тасование.

Следующие упражнения описывают варианты быстрой сортировки. Каждый из них требует реализации, но естественно также использовать программу `SortCompare` для экспериментальной оценки эффективности каждой предложенной модификации.

2.3.17. *Сигнальные ключи.* Уберите из внутреннего цикла `while` в коде алгоритма 2.5 обе проверки на выход за границы массива. Проверка на выход за левую границу массива излишня, т.к. центральный элемент выступает в качестве сигнального маркера (`v` не может быть меньше, чем `a[lo]`). Чтобы удалить другую проверку, нужно сразу после тасования поместить значение ключа, максимального во всем массиве, в элемент `a[length-1]`. Этот элемент не будет перемещаться (разве что обменяется с другим таким же ключом) и будет служить сигнальным маркером во всех подмассивах с конца массива. *Примечание*: при сортировке внутренних подмассивов левый элемент в правом подмассиве служит сигнальным маркером для правого конца подмассива.

2.3.18. *Разбиение по медиане из трех.* Добавьте в быструю сортировку разбиение по медиане из трех элементов, как описано в тексте (см. подраздел "Разбиение по медиане из трех"). С помощью тестов с удвоением определите эффективность этого добавления.

2.3.19. *Разбиение по медиане из пяти.* Реализуйте быструю сортировку на основе разбиения по медиане из случайной выборки пяти элементов из подмассива. Разме-

щайте элементы из выборки по соответствующим концам массива, чтобы в разбиении участвовала только медиана. С помощью тестов с удвоением определите эффективность этого добавления по сравнению со стандартным алгоритмом и разбиением по медиане из трех (см. предыдущее упражнение). *Дополнительно*: придумайте алгоритм с медианой из пяти, который использует менее семи сравнений для любых данных.

2.3.20. *Нерекурсивная быстрая сортировка.* Реализуйте нерекурсивную версию быстрой сортировки с главным циклом, где подмассивы выбираются для разбиения из стека, а результирующие подмассивы заталкиваются в стек. *Примечание*: сначала заталкивайте в стек больший из подмассивов: это гарантирует, что в стеке будет не больше $\lg N$ элементов.

2.3.21. *Нижняя граница для сортировки с одинаковыми ключами.* Завершите первую часть доказательства утверждения Н, используя логику из доказательства утверждения И, а также наблюдение, что существуют $N! \,/\, f_1!\, f_2! \ldots f_k!$ различных способов расставить ключи с k различными значениями, где i-е значение появляется с частотой f_i ($= Np_i$ в обозначениях утверждения Н) и $f_1 + \ldots + f_k = N$.

2.3.22. *Быстрое трехчастное разбиение* (Дж. Бентли и Д. Мак-Илрой, рис. 2.3.9). Реализуйте сортировку с оптимальной энтропией, при которой элементы с равными ключами находятся на левом и правом концах подмассива. Используйте индексы `p` и `q` такие, что все `a[lo..p-1]` и `a[q+1..hi]` равны `a[lo]`, индекс `i` такой, что `a[p..i-1]` меньше `a[lo]`, и индекс `j` такой, что все `a[j+1..q]` больше `a[lo]`. Добавьте код во внутренний цикл разбиения для обмена `a[i]` с `a[p]` (и увеличения `p`), если оно равно `v`, перед выполнением обычных сравнений `a[i]` и `a[j]` с `v`. После завершения цикла разбиения добавьте код для перемещения элементов с одинаковыми ключами на свои места. *Примечание*: Этот код дополняет код, приведенный в тексте, т.е. он выполняет дополнительные обмены для ключей, равных ключу центрального элемента, а код в тексте выполняет дополнительные обмены для ключей, которые *не* равны ключу центрального элемента.

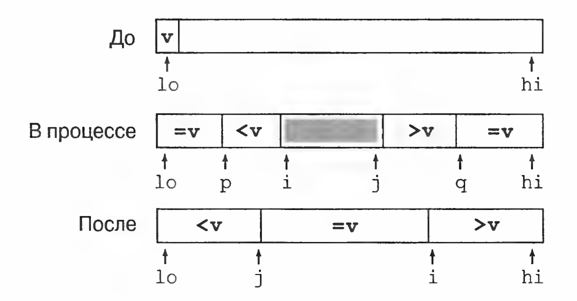


Рис. 2.3.9. *Трехчастное разбиение Бентли-Макилроя*

2.3.23. *Системная сортировка в Java.* Добавьте в реализацию из упражнения 2.3.22 код *девятки Тьюки* (Tukey ninther): выбираются три набора по три элемента, в каждом находится медиана, и в качестве центрального элемента берется медиана из этих трех медиан. Кроме того, добавьте отсечку для перехода на сортировку вставками для небольших подмассивов.

2.3.24. *Сортировка с выборкой* ((У. Фрезер) W. Frazer и (Э. Мак-Келлар) A. McKellar). Реализуйте быструю сортировку с использованием выборки размером 2^k-1. Сначала отсортируйте выборку, потом выполните рекурсивное разбиение по медиане этой выборки и распределение двух половин оставшихся элементов выборки в каждый из подмассивов, чтобы задействовать их в подмассивах без повторной сортировки. Этот алгоритм называется *сортировкой с выборкой* (samplesort).

Эксперименты

2.3.25. *Сортировка с отсечкой на вставки.* Реализуйте быструю сортировку с отсечкой на сортировку вставками для подмассивов с менее чем M элементами, и эмпирически определите значение M, для которого в вашей вычислительной среде быстрая сортировка быстрее всего упорядочивает случайные массивы из N значений `double`, для $N = 10^3$, 10^4, 10^5 и 10^6. Начертите график средних времен выполнения для каждого M от 0 до 30. *Примечание*: в алгоритм 2.2 нужно добавить метод `sort()` с тремя аргументами — такой, что вызов `Insertion.sort(a, lo, hi)` сортирует подмассив `a[lo..hi]`.

2.3.26. *Размеры подмассивов.* Напишите программу, которая выводит гистограмму размеров подмассивов, оставляемых для сортировки вставками при упорядочении быстрой сортировкой массива размером N с отсечкой подмассивов размером менее M. Выполните программу для $M = 10$, 20 и 50 и $N = 10^5$.

2.3.27. *Игнорирование небольших подмассивов.* Экспериментально сравните подход, описанный в упражнении 2.3.25, со следующей стратегией для обработки небольших подмассивов. В быстрой сортировке просто игнорируйте небольшие подмассивы, а потом, после завершения быстрой сортировки, выполните одну сортировку вставками. *Примечание*: с помощью этих экспериментов можно оценить размер кеша памяти в вашем компьютере, поскольку, когда массив не будет входить в кеш, производительность этого метода заметно снизится.

2.3.28. *Глубина рекурсии.* Экспериментально определите среднюю рекурсивную глубину, используемую быстрой сортировкой с отсечкой для массивов размером M, при сортировке массивов N различных элементов, для $M = 10$, 20 и 50 и $N = 10^3$, 10^4, 10^5 и 10^6.

2.3.29. *Рандомизация.* Эмпирически сравните эффективность стратегии выбора случайного центрального элемента со стратегией предварительного тасования массива (как в тексте). Используйте отсечку для массивов размером M и сортируйте массивы N различных элементов, для $M = 10$, 20 и 50 и $N = 10^3$, 10^4, 10^5 и 10^6.

2.3.30. *Крайние случаи.* Проверьте работу быстрой сортировки на больших неслучайных массивах, которые описаны в упражнениях 2.1.35 и 2.1.36 — с предварительным тасованием и без него.

2.3.31. *Гистограмма значений времени выполнения.* Напишите программу, которая принимает из командной строки аргументы N и T, выполняет T раз быструю сортировку массива N случайных значений `Double` и выводит гистограмму полученных времен выполнения. Выполните программу для $N = 10^3$, 10^4, 10^5 и 10^6 и настолько большого T, чтобы кривые выглядели гладко. Главная трудность в данном упражнении — подходящее масштабирование экспериментальных результатов.

2.4. Очереди с приоритетами

Во многих приложениях нужно обрабатывать элементы с ключами по порядку, но не обязательно полностью по порядку и не обязательно все сразу. Часто собирается коллекция элементов, потом обрабатывается один из них с наибольшим ключом, затем, возможно, собираются еще элементы, потом обрабатывается элемент с текущим наибольшим ключом и т.д. У вас наверняка есть компьютер (или смартфон), который может одновременно выполнять несколько приложений. Для этого событиям, связанным с этими приложениями, назначаются приоритеты, и для обработки выбирается следующее событие с максимальным приоритетом. Например, в большинстве смартфонов входящим звонкам назначается больший приоритет, чем играм.

Подходящий тип данных для такой среды поддерживает две операции: *извлечь наибольший* и *вставить*. Такой тип данных называется *очередь с приоритетами* (priority queue). Использование очередей с приоритетами похоже на использование очередей (извлекается самый старый элемент) и стеков (извлекается самый свежий), но эффективно реализовать их гораздо сложнее.

В настоящем разделе, после краткого знакомства с элементарными представлениями, где одна или обе операции выполняются за линейное время, мы рассмотрим классическую реализацию очереди с приоритетами на основе структуры данных *бинарная пирамида*, где элементы хранятся в массиве с определенными ограничениями, которые позволяют эффективные (за логарифмическое время) реализации операций *извлечь наибольший* и *вставить*.

Вот некоторые важные области применения очередей с приоритетами: системы моделирования, где ключи соответствуют моментам событий, которые должны обрабатываться в хронологическом порядке; планировка работ, где ключи соответствуют приоритетам, указывающим, какие задачи следует обрабатывать в первую очередь; численные расчеты, где ключи представляют собой погрешности вычислений и определяют порядок, в котором их следует обрабатывать. В главе 6 мы рассмотрим подробный учебный пример с применением очередей с приоритетами для моделирования столкновений элементарных частиц.

На основе любой очереди с приоритетами можно построить сортирующий алгоритм, который вставляет в очередь последовательность элементов, а затем извлекает из нее текущие наименьшие элементы. Важный алгоритм сортировки — *пирамидальная сортировка* — также естественно вытекает из наших реализаций очереди с приоритетами на основе пирамидальной структуры. Далее в этой книге мы увидим, как использовать очереди с приоритетами в качестве строительных блоков для других алгоритмов. В главе 4 будет показано, что очереди с приоритетами являются удобной абстракцией для реализации нескольких фундаментальных алгоритмов поиска на графах, а в главе 5 мы разработаем алгоритм сжатия данных, который будет использовать методы из данного раздела. Это лишь несколько примеров важной роли, которую играют очереди с приоритетами в качестве средства проектирования алгоритмов.

API-интерфейс

Очередь с приоритетами представляет собой прототипный *абстрактный тип данных* (см. раздел 1.2): она содержит набор значений и операций над этими значениями и предоставляет удобную абстракцию, которая позволяет отделить прикладные программы (клиенты) от различных реализаций, которые мы рассмотрим в данном разделе. Как и в разделе 1.2, мы точно определяем операции, формулируя *API-интерфейс*, который предоставляет клиентам нужную им информацию (рис. 2.4.1). Очереди с приоритетами характеризуются операциями *извлечь наибольший* и *вставить*, поэтому мы будем рассматривать в первую очередь их. Мы выбрали имя delMax() для метода извлечения наибольшего элемента и insert() для метода вставки. Сравнение ключей мы будем выполнять только вспомогательным методом less() — как обычно для сортировки. Поэтому при наличии повторяющихся ключей *наибольший* элемент означает *любой* элемент с наибольшим значением ключа. Для полноты в API-интерфейс необходимо добавить конструкторы (такие как конструкторы для стеков и очередей) и операцию *проверить, пусто ли*. Для повышения гибкости мы используем обобщенную реализацию с параметризованным типом Key, который должен реализовать интерфейс Comparable. Такой выбор снимает различие между элементами и ключами и делает описания структур данных и алгоритмов более понятными и компактными. Например, мы будем говорить "больший ключ", а не "больший элемент" или "элемент с большим ключом".

Для удобства кодирования клиентов API-интерфейс содержит три конструктора, которые позволяют создавать очереди с приоритетами с первоначально заданным фиксированным размером (и, возможно, инициализированные заданным массивом ключей). Чтобы код был понятнее, мы будем использовать, где это уместно, отдельный класс MinPQ — такой же, как MaxPQ, но содержащий метод delMin(), который извлекает и возвращает наименьший ключ в очереди. Любую реализацию MaxPQ несложно преобразовать в реализацию MinPQ и наоборот, просто изменив смысл сравнения в функции less().

public class **MaxPQ**<Key extends Comparable<Key>>

	MaxPQ()	*создание очереди с приоритетами*
	MaxPQ(int max)	*создание очереди с приоритетами первоначальным объемом* max
	MaxPQ(Key[] a)	*создание очереди с приоритетами из ключей в массиве* a[]
void	insert(Key v)	*вставка ключа в очередь с приоритетами*
Key	max()	*возврат наибольшего ключа*
Key	delMax()	*возврат и извлечение наибольшего ключа*
boolean	isEmpty()	*пуста ли данная очередь с приоритетами?*
int	size()	*количество ключей в очереди с приоритетами*

Рис. 2.4.1. *API-интерфейс для обобщенной очереди с приоритетами*

Клиент очереди с приоритетами

Чтобы лучше понять ценность абстракции очереди с приоритетами, рассмотрим следующую задачу. Пусть имеется большой входной поток из N строк и связанных с ними целых значений, и в этом потоке нужно найти M наибольших или наименьших чисел (и соответствующих строк). Такой поток может содержать, например, финансовые транзакции, среди которых необходимо найти самые крупные, данные о содержании пес-

тицидов в сельскохозяйственной продукции, среди которых нужно найти наименьшие, результаты научного эксперимента или много чего еще. В некоторых приложениях размер входного потока настолько велик, что его проще считать неограниченным. Один из способов решения такой задачи — сортировка входного потока и выборка M наибольших ключей, но, как было только что сказано, входной поток слишком велик для этого. Другой подход — сравнение каждого нового ключа с M найденными к данному моменту наибольшими, но и его стоимость слишком велика, если M не очень мало. Очереди с приоритетами позволяют решить эту задачу с помощью клиента TopM класса MinPQ, приведенного в листинге 2.4.1 — *если* можно разработать эффективные реализации и для insert(), и для delMin(). А это и есть наша цель в данном разделе. Для больших значений N, которые могут встречаться в современной вычислительной инфраструктуре, эти реализации могут означать разницу между возможностью решить задачу и отсутствием ресурсов для этого.

Таблица 2.4.1. Стоимости поиска *M* наибольших элементов в потоке из *N* элементов

Клиент	Порядок роста	
	Время	Память
Клиент сортировки	$N \log N$	N
Клиент очереди с приоритетами с примитивной реализацией	NM	M
Клиент очереди с приоритетами с пирамидальной реализацией	$N \log M$	M

Листинг 2.4.1. Клиент очереди с приоритетами

```
public class  TopM
{
   public static void main(String[] args)
   {  // Вывод максимальных M строк из входного потока.
      int M = Integer.parseInt(args[0]);
      MinPQ<Transaction> pq = new MinPQ<Transaction>(M+1);
      while (StdIn.hasNextLine())
      {  // Создание элемента из текстовой строки и занесение ее в ОсП.
         pq.insert(new Transaction(StdIn.readLine()));
         if (pq.size() > M)
            pq.delMin();  // Если в ОсП стало M+1 элементов — удаление наименьшего.
      }  // В ОсП находятся M верхних элементов.
      Stack<Transaction> stack = new Stack<Transaction>();
      while (!pq.isEmpty()) stack.push(pq.delMin());
      for (Transaction t : stack) StdOut.println(t);
   }
}
```

Для заданного в командной строке целого значения M и входного потока, где каждая строка содержит транзакцию, этот клиент класса MinPQ выводит M строк с максимальными числами. Для создания очереди с приоритетами используется класс Transaction (рис. 1.2.10, упражнение 1.2.19 и упражнение 2.1.21), в качестве ключей берутся числа, и когда размер очереди с приоритетами достигает M, то после каждой вставки удаляется минимальное число. После обработки всех транзакций наибольшие M должны выбираться из очереди в порядке возрастания, поэтому код сначала помещает их в стек, а затем просматривает стек, чтобы изменить порядок и вывести транзакции в порядке увеличения.

```
% more tinyBatch.txt
Turing          6/17/1990    644.08
vonNeumann      3/26/2002   4121.85
Dijkstra        8/22/2007   2678.40
vonNeumann      1/11/1999   4409.74
Dijkstra       11/18/1995    837.42
Hoare           5/10/1993   3229.27
vonNeumann      2/12/1994   4732.35
Hoare           8/18/1992   4381.21
Turing          1/11/2002     66.10
Thompson        2/27/2000   4747.08
Turing          2/11/1991   2156.86
Hoare           8/12/2003   1025.70
vonNeumann     10/13/1993   2520.97
Dijkstra        9/10/2000    708.95
Turing         10/12/1993   3532.36
Hoare           2/10/2005   4050.20
```

```
% java TopM 5 < tinyBatch.txt
Thompson        2/27/2000   4747.08
vonNeumann      2/12/1994   4732.35
vonNeumann      1/11/1999   4409.74
Hoare           8/18/1992   4381.21
vonNeumann      3/26/2002   4121.85
```

Элементарные реализации

Для реализации очередей с приоритетами можно выбрать одну из четырех базовых структур данных, рассмотренных в главе 1. Мы можем использовать массив или связный список, упорядоченный либо неупорядоченный. Эти реализации удобны для небольших очередей с приоритетами, ситуаций, где одна из операций доминирует по трудоемкости, или если имеются какие-то априорные сведения об упорядоченности поступающих ключей. В силу элементарности реализаций мы лишь кратко опишем их в тексте, но код оставим на самостоятельную проработку (см. упражнение 2.4.3).

Представление массивом (неупорядоченное)

Пожалуй, самая простая реализация очереди с приоритетами — на основе нашего кода для стека из раздела 1.3. Код операции *вставить* совпадает с кодом операции *затолкнуть* для стека. А для реализации операции *извлечь наибольший* можно добавить код, похожий на внутренний цикл сортировки выбором, где наибольший элемент обменивается с последним элементом, а затем удаляется, как в операции `pop()` для стеков. Так же, как и для стеков, можно добавить код изменения размера массива, чтобы структура данных всегда занимала не менее четверти выделенной памяти и не выходила за ее пределы.

Представление массивом (упорядоченное)

Другой способ — добавить в код операции *вставить* сдвиг больших элементов на одну позицию вправо, чтобы ключи в массиве были упорядочены (как в сортировке вставками). Тогда наибольшие элементы будут всегда находиться в конце массива, а код операции *извлечь наибольший* будет совпадать с операцией *вытолкнуть* для стека.

Представления связным списком

Аналогично можно взять за основу код для стека в виде связного списка и изменить либо код операции `pop()` для поиска и возврата максимального элемента, либо код операции `push()`, чтобы поддерживать *обратную* упорядоченность, и `pop()` для отстегивания от списка и возврата первого (максимального) элемента списка.

Таблица 2.4.2. Порядок роста времени выполнения в худшем случае для реализаций очереди с приоритетами

Структура данных	Вставка	Извлечение наибольшего
Упорядоченный массив	N	1
Неупорядоченный массив	1	N
Пирамида	$\log N$	$\log N$
Невозможно	1	1

Использование неупорядоченных последовательностей является прототипом *ленивого* подхода к решению задачи: мы откладываем выполнение каких-то действий до того момента, когда понадобится их результат (наибольшее значение); применение упорядоченных последовательностей является прототипом *энергичного* подхода к решению задачи: мы выполняем максимальный объем работы заранее (поддерживаем упорядоченность списка при вставках), чтобы в последующем операции выполнялись эффективно.

Существенное различие между реализациями стеков или очередей и реализациями очередей с приоритетами — в производительности. Для стеков и очередей мы смогли разработать реализации всех операций, требующие *постоянного* времени, но для очередей с приоритетами все рассмотренные элементарные реализации требуют в худшем случае *линейного* времени либо для операции *вставить*, либо для операции *извлечь наибольший*. Пирамидальная структура данных, которую мы рассмотрим ниже, позволяет реализации, в которых *обе* операции выполняются гарантированно быстро.

Таблица 2.4.3. Последовательность операций над очередью с приоритетами

Операция	Аргумент	Возвр. значение	Размер	Содержимое (неупорядоченное)	Содержимое (упорядоченное)
вставить	P		1	**P**	**P**
вставить	Q		2	P **Q**	P **Q**
вставить	E		3	P Q **E**	**E** P Q
извлечь наибольший		**Q**	2	P E	E P
вставить	X		3	P E **X**	E P **X**
вставить	A		4	P E X **A**	**A** E P X
вставить	M		5	P E X A **M**	A E **M** P X
извлечь наибольший		**X**	4	P E M A	A E M P
вставить	P		5	P E M A **P**	A E M P **P**
вставить	L		6	P E M A P **L**	A E **L** M P P
вставить	E		7	P E M A P L **E**	A E **E** L M P P
извлечь наибольший		**P**	6	E E M A P L	A E E L M P

10 Зak. 3239

Определения пирамиды

Бинарная пирамида (binary heap)[1] — это структура данных, которая может эффективно поддерживать базовые операции очереди с приоритетами. В бинарной пирамиде ключи хранятся в массиве так, что каждый ключ обязательно больше (или равен) ключей в двух других конкретных позициях. А каждый из этих ключей должен быть больше (или равен) ключей двух других ключей и т.д. Это упорядочение легко увидеть, если представлять ключи в виде структуры бинарного дерева, где каждый ключ связан с другими ключами, меньшими его.

Определение. Бинарное дерево называется *пирамидально упорядоченным*, если ключ в каждом узле больше или равен ключам в двух дочерних узлах этого узла (если они есть).

Соответственно, ключ в каждом узле пирамидально упорядоченного бинарного дерева меньше или равен ключу в родительском узле этого узла (если он есть). При движении вверх от любого узла получается неубывающая последовательность ключей, а при движении вниз — невозрастающая последовательность.

Утверждение П. Наибольший ключ в пирамидально упорядоченном бинарном дереве находится в его корне.

Доказательство. По индукции по размеру дерева.

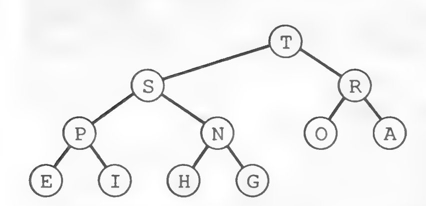


Рис. 2.4.2. Пирамидально упорядоченное полное бинарное дерево

Представление двоичной пирамиды

Если для пирамидально упорядоченных бинарных деревьев использовать связное представление, то с каждым ключом придется ассоциировать три ссылки, чтобы можно было перемещаться по дереву вверх и вниз (каждый узел должен содержать указатель на родительский узел и на два дочерних). Вместо этого гораздо удобнее использовать *полное* бинарное дерево, вроде приведенного на рис. 2.4.2. Чтобы нарисовать такую структуру, нужно начать с корня дерева, а затем продолжать вниз и слева направо, присоединяя к каждому узлу по два узла уровнем ниже, пока не будут нарисованы все N узлов. Полные деревья позволяют применять компактное "массивное" представление, в котором не используются явные ссылки. А именно, полные бинарные деревья представляются в массиве с помощью размещения узлов в *поуровневом порядке* (level order): корень находится в позиции 1, его дочерние узлы — в позициях 2 и 3, их дочерние узлы — в позициях 4, 5, 6 и 7 и т.д.

---

[1] В этой области нет устоявшейся удачной русскоязычной терминологии, и термин *heap* переводится и как "куча", и как "пирамида", и как "пирамидальное дерево", и даже как "сортирующее дерево". Однако "куча" обычно используется в контексте распределения памяти — системой или другой средой времени выполнения, а из остальных вариантов выбран самый краткий и согласованный с другими терминами. Да и смысл этой структуры — вынесение наверх максимальных элементов — немного похож на "финансовые пирамиды", где обитатели высших слоев получают больше тех, кто находится ниже. — *Прим. перев.*

Определение. *Бинарная пирамида* — это коллекция ключей, упорядоченных в виде полного пирамидально упорядоченного бинарного дерева, которое размещено в массиве в поуровневом порядке (без использования первого элемента).

Для краткости с данного момента мы будем опускать термин "бинарная" и применять просто слово *пирамида*, имея в виду бинарную пирамиду. В пирамиде родитель узла в позиции k находится в позиции $\lfloor k/2 \rfloor$ и, наоборот, два потомка узла в позиции k находятся в позициях $2k$ и $2k+1$. Вместо применения явных ссылок (как в структурах бинарных деревьев, которые будут рассматриваться в главе 3) можно перемещаться вверх и вниз, выполняя простые арифметические действия над индексами массива: для перемещения *вверх* по дереву от элемента `a[k]` нужно заменить `k` значением `k/2`, а для перемещения *вниз* по дереву — заменить `k` на значение `2*k` или `2*k+1` (рис. 2.4.3).

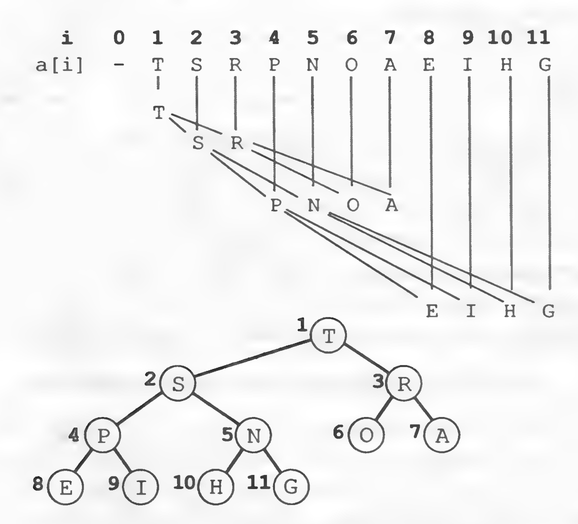


Рис. 2.4.3. Представление пирамиды

Полные бинарные деревья, представляемые в виде массивов (пирамид), являются жесткими структурами, но их гибкости все-таки хватает, чтобы реализовывать с их помощью эффективные операции очередей с приоритетами. А именно, мы будем использовать их для разработки реализаций операций *вставить* и *извлечь наибольший* с логарифмическим временем выполнения. Эти алгоритмы опираются на возможность перемещаться вверх и вниз по дереву без задействования указателей и имеют гарантированно логарифмическую производительность — в силу следующего свойства полных бинарных деревьев.

Утверждение Р. Высота полного бинарного дерева размером N равна $\lfloor \lg N \rfloor$.

Доказательство. Этот результат нетрудно доказать методом индукции или заметив, что высота бинарного дерева увеличивается на 1 каждый раз, когда N равно степени 2.

Алгоритмы работы с пирамидами

Мы будем представлять пирамиду размером N в приватном массиве `pq[]` длиной $N+1$: элемент `pq[0]` не используется, а пирамида расположена в элементах с `pq[1]` по `pq[N]`. Как и в случае алгоритмов сортировки, мы будем обращаться к клю-

чам только с помощью приватных вспомогательных функций less() и exch() (см. листинг 2.4.2), но, раз уж все элементы находятся в переменной экземпляров pq[], мы будем использовать более компактные реализации (см. листинги 2.4.3 и 2.4.4), в которых не требуется передавать имя массива в качестве параметра. Рассматриваемые нами операции над пирамидами работают так: сначала выполняется простое изменение, которое может нарушить условие пирамидальности, а затем осуществляется проход по пирамиде с ее коррекцией для восстановления статус-кво. Это процесс мы назовем *восстановлением пирамидальности* или *восстановлением пирамидального порядка*.

Листинг 2.4.2. Методы сравнения и перестановки для реализаций пирамиды

```
private boolean less(int i, int j)
{  return pq[i].compareTo(pq[j]) < 0;  }

private void exch(int i, int j)
{  Key t = pq[i]; pq[i] = pq[j]; pq[j] = t;  }
```

Здесь могут быть два варианта. Если приоритет какого-то узла увеличился (или в нижний слой пирамиды добавлен новый узел), то для восстановления пирамидального порядка необходимо пройти *вверх* по пирамиде. А если приоритет какого-то узла уменьшился (например, если корневой узел заменен другим, с меньшим ключом), то для восстановления пирамидального порядка понадобится пройти *вниз* по пирамиде. Сначала мы разберемся, как реализовать эти две базовые вспомогательные операции, а затем посмотрим, как с их помощью реализовать операции *вставить* и *извлечь наибольший*.

Восходящее восстановление пирамидальности (всплытие)

Если пирамидальный порядок нарушен из-за того, что ключ какого-то узла стал *больше*, чем ключ его родительского узла, то такое нарушение можно исправить, обменяв этот узел с его родителем. После обмена узел будут больше, чем его дочерние узлы (один из них — бывший родитель, а другой меньше бывшего родителя, потому что он был его потомком), но этот узел все равно может быть больше нового родителя. Это нарушение можно исправить точно так же — и узел-нарушитель будет перемещаться вверх по пирамиде, пока не столкнется с узлом с большим ключом или станет корнем (рис. 2.4.4). Кодирование этого процесса не составляет труда, если вспомнить, что родитель узла в позиции k находится в позиции k/2. Цикл в методе swim() сохраняет свойство, что единственное место, где может быть нарушен пирамидальный порядок — это узел в позиции k, который, возможно, больше своего родителя. Поэтому когда цикл доходит до состояния, где этот узел уже не больше своего родителя, пирамидальный порядок присутствует во всей структуре. В соответствии с именем метода (swim — плавать), узел с большим ключом как бы *всплывает* на более высокие уровни пирамиды.

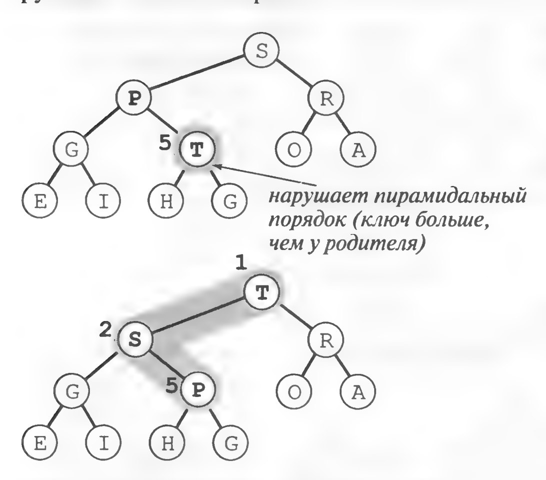


Рис. 2.4.4. Восходящее восстановление пирамидальности (всплытие)

Листинг 2.4.3. Реализация восходящего восстановления пирамидальности (всплытия)

```
private void swim(int k)
{
   while (k > 1 && less(k/2, k))
   {
      exch(k/2, k);
      k = k/2;
   }
}
```

Нисходящее восстановление пирамидальности (погружение)

Если пирамидальный порядок нарушен из-за того, что ключ какого-то узла стал *меньше*, чем один или оба ключа его дочерних узлов, то такое нарушение можно исправить, обменяв этот узел с *большим* из его предков. Этот обмен может привести к нарушению пирамидальности в дочернем узле, которое исправляется точно так же — и узел-нарушитель будет перемещаться вниз по пирамиде, пока не столкнется с двумя меньшими (или равными) потомками или дойдет до основания (рис. 2.4.5). Код также непосредственно следует из того факта, что потомки узла в позиции k находятся в позициях 2k и 2k+1. В соответствии с именем метода (sink — тонуть), узел с маленьким ключом как бы *погружается* на более низкие уровни пирамиды.

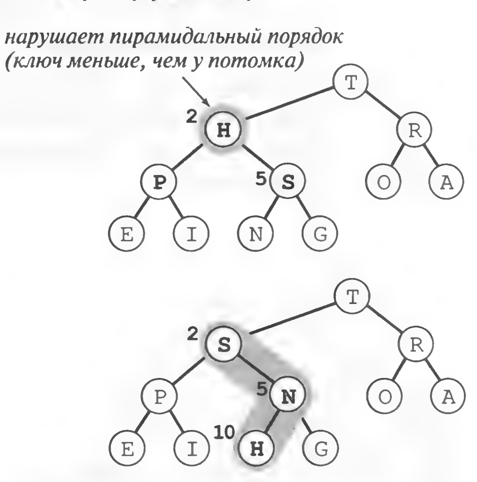


Рис. 2.4.5. *Нисходящее восстановление пирамидальности (погружение)*

Листинг 2.4.4. Реализация нисходящего восстановления пирамидальности (погружения)

```
private void sink(int k)
{
   while (2*k <= N)
   {
      int j = 2*k;
      if (j < N && less(j, j+1)) j++;
      if (!less(k, j)) break;
      exch(k, j);
      k = j;
   }
}
```

Если представить пирамиду как отображение корпоративной иерархии, где дочерние узлы представляют собой подчиненных, а родительский узел — их непосредственного руководителя, то этим операциям можно приписать забавную интерпретацию. Операция swim() соответствует только что нанятому способному новому менеджеру, который повышается по цепочке субординации (обмениваясь должностями с менее квалифицированными начальниками), пока не доберется до более квалифицированного босса.

Операция `sink()` аналогична ситуации, когда президент компании увольняется и заменяется кем-то извне корпорации. Если наиболее компетентный из его подчиненных способнее его, они обмениваются должностями, и этот процесс продолжается по цепочке субординации, понижая в должности вновь прибывшего и повышая подчиненных, пока оба подчиненных не окажутся менее компетентными. Такие идеальные ситуации редко встречаются в реальном мире, но они помогают понять работу базовых операций с пирамидами.

Эти операции `sink()` и `swim()` лежат в основе эффективной реализации API-интерфейса очереди с приоритетами, как показано на рис. 2.4.6 и в листинге 2.4.5.

Вставить. Новый ключ добавляется в конец массива, размер пирамиды увеличивается, а затем добавленный ключ всплывает в структуре данных, пока не будет восстановлена ее пирамидальность.

Извлечь наибольший. Из начала массива выбирается наибольший ключ, на его место заносится элемент из конца массива, размер пирамиды уменьшается, а затем переставленный ключ погружается в структуре данных, пока не будет восстановлена ее пирамидальность.

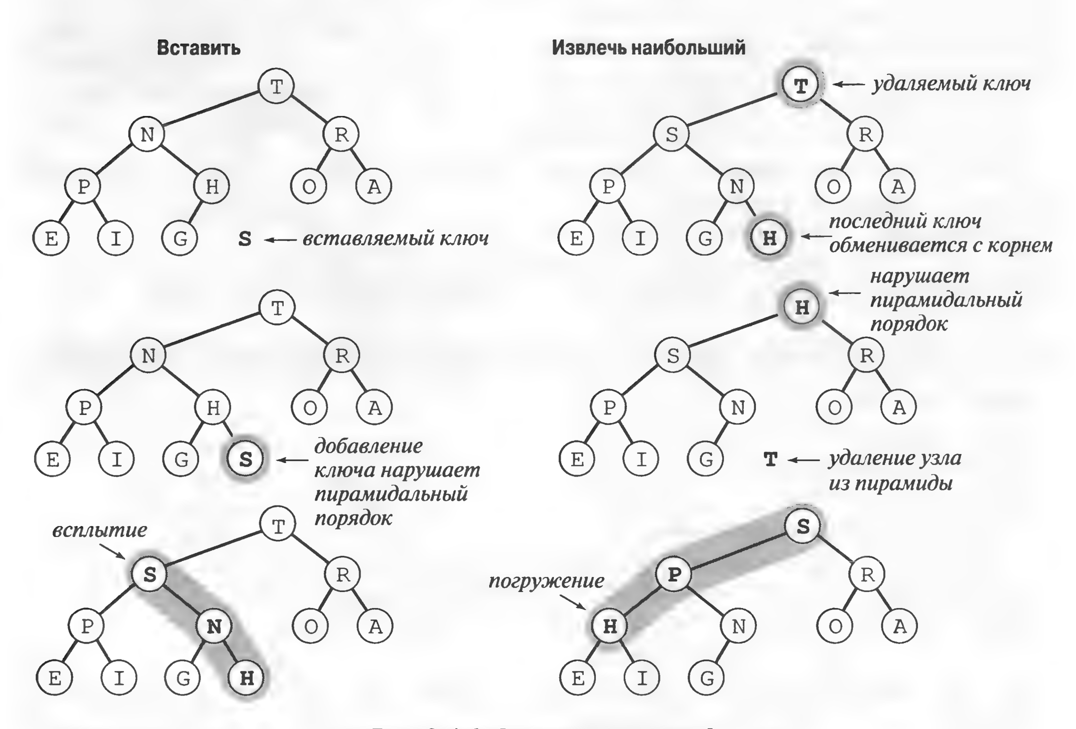


Рис. 2.4.6. Операции в пирамиде

Листинг 2.4.5. Алгоритм 2.6. Очередь с приоритетами на основе пирамиды

```
public class MaxPQ<Key extends Comparable<Key>>
{
  private Key[] pq;    // пирамидально упорядоченное полное бинарное дерево
  private int N = 0;   //    в элементах pq[1..N] (pq[0] не используется)

  public MaxPQ(int maxN)
  { pq = (Key[]) new Comparable[maxN+1]; }

  public boolean isEmpty()
  { return N == 0; }

  public int size()
  { return N; }

  public void insert(Key v)
  {
    pq[++N] = v;
    swim(N);
  }

  public Key delMax()
  {
    Key max = pq[1];          // Извлечение максимального ключа из корня.
    exch(1, N--);             // Обмен с последним элементом.
    pq[N+1] = null;           // Чтобы не было праздных ссылок.
    sink(1);                  // Восстановление пирамидальности.
    return max;
  }

  // Реализации этих вспомогательных методов см. в листингах 2.1.1 и 2.1.3.
  private boolean less(int i, int j)
  private void exch(int i, int j)
  private void swim(int k)
  private void sink(int k)
}
```

Очередь с приоритетами хранится в пирамидально упорядоченном полном бинарном дереве — в массиве pq[], где pq[0] не используется, а N ключей находятся в элементах с pq[1] по pq[N]. В реализации метода insert() увеличивается значение N, в конец добавляется новый элемент, а затем вызывается метод swim(), который восстанавливает пирамидальный порядок. В реализации метода delMax() возвращаемое значение выбирается из pq[1], потом pq[N] переносится в pq[1], размер пирамиды уменьшается, и вызывается метод sink(), который восстанавливает пирамидальный порядок. В уже ненужную позицию pq[N+1] записывается null, чтобы система могла изъять память, связанную с ней. Код для динамического изменения размера массива, как обычно, не приведен (см. раздел 1.3). Другие конструкторы см. в упражнении 2.4.19.

Алгоритм 2.6 решает базовую задачу, которая была поставлена в начале данного раздела: это реализация API-интерфейса очереди с приоритетами, в которой и операция *вставить*, и операция *извлечь наибольший* гарантированно выполняются за логарифмическое время относительно размера очереди.

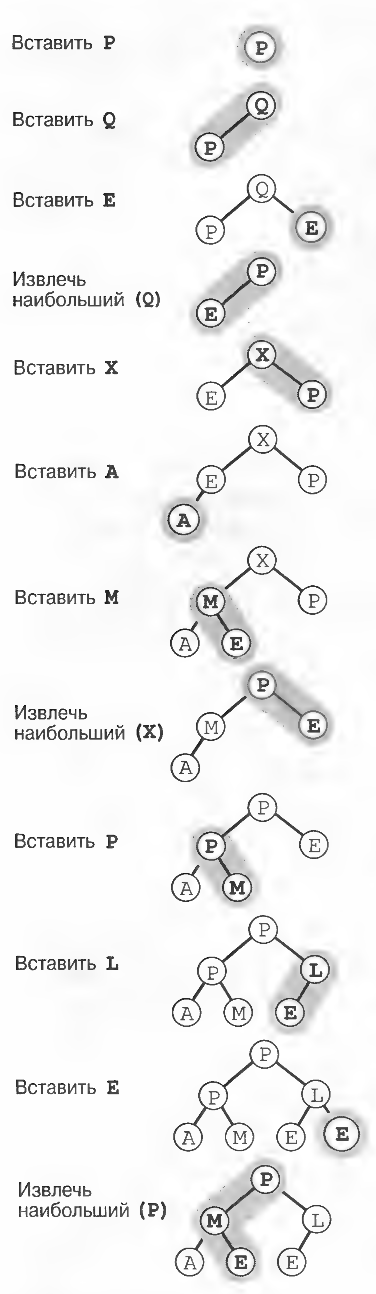


Рис. 2.4.7. Операции над очередью с приоритетами в пирамиде

Утверждение С. В очереди с приоритетами объемом N ключей пирамидальные алгоритмы выполняют не более $1 + \lg N$ сравнений для операции *вставить*, и не более $2\lg N$ сравнений для операции *извлечь наибольший*.

Доказательство. Согласно утверждению P, обе операции выполняют перемещение вдоль пути между корнем и основанием пирамиды, а число ссылок в этом пути не больше $\lg N$. Операция *извлечь наибольший* требует двух сравнений для каждого узла на пути (кроме самого нижнего): один — чтобы найти дочерний узел с большим ключом, а другой — чтобы решить, нужно ли повысить этот узел.

В типичных приложениях, в которых необходимо выполнять беспорядочную смесь операций вставки и извлечения наибольшего значения (как на рис. 2.4.7), утверждение С представляет важный прорыв в производительности, который резюмирован в табл. 2.4.2. Там, где элементарные реализации на основе упорядоченного или неупорядоченного массива требуют линейного времени на выполнение одной из операций, реализация на основе пирамидальной структуры гарантированно выполняет обе операции за логарифмическое время. Это усовершенствование означает различие между возможностью и полной невозможностью решить задачу.

Многочастные пирамиды

Нетрудно изменить наш код так, чтобы создавать пирамиды, основанные на представлении полных пирамидально упорядоченных *тернарных* деревьев в виде массивов. В таких деревьях элемент в позиции k больше или равен элементам в позициях $3k-1$, $3k$ и $3k+1$, и меньше или равен элементу в позиции $\lfloor (k+1)/3 \rfloor$, для всех индексов от 1 до N. Ненамного сложнее использовать и d-арные пирамиды для любого заданного d. Здесь нужно следить за соотношением меньшей стоимости из-за снижения высоты дерева ($\log_d N$) и большей стоимости поиска наибольшего из d потомков для каждого узла. Это соотношение зависит от деталей реализации и ожидаемой относительной частоты операций.

Изменение размера массива

В нашу реализацию можно добавить конструктор без аргументов, код для удвоения размера массива в методе `insert()` и код для сокращения размера массива вдвое в методе `delMax()` — в точности так же, как для стеков в разделе 1.3. Таким образом, клиентам нет необходимости заботиться об ограничениях на размер массива. В случае произвольного размера очереди с приоритетами и изменения

размеров массива логарифмические временные границы, о которых сказано в утверждении С, являются *амортизированными* (см. упражнение 2.4.22).

Неизменность ключей

Очередь с приоритетами содержит объекты, которые создаются клиентами, но при этом подразумевается, что клиентский код не изменяет ключи (что может привести к нарушению пирамидальной упорядоченности). Можно разработать механизмы, которые будут следить за этим, но программисты обычно так не делают — ведь это усложняет код и может снизить производительность.

Индексная очередь с приоритетами

Во многих случаях имеет смысл разрешить клиентам доступ к элементам, которые уже занесены в очередь с приоритетами. Один из способов легко сделать это — связать с каждым элементом уникальный целочисленный *индекс*. Более того, часто бывает так, что клиенты работают с множеством элементов известного размера *N* и, возможно, уже используют (параллельно) массивы для хранения информации об этих элементах — поэтому другой клиентский код может уже применять целочисленные индексы для обращения к элементам. Эти соображения приводят к API-интерфейсу, приведенному на рис. 2.4.8.

public class **IndexMinPQ**<Item extends Comparable<Item>>		
	IndexMinPQ(int maxN)	*создание очереди с приоритетами емкостью* maxN *с возможными индексами от* 0 *до* maxN-1
void	insert(int k, Item item)	*вставка элемента* item *и связывание его с* k
void	change(int k, Item item)	*замена значения элемента, связанного с* k, *на* item
boolean	contains(int k)	*связан ли индекс* k *с каким-нибудь элементом?*
void	delete(int k)	*удаление* k *и связанного с ним элемента*
Item	min()	*возврат минимального элемента*
int	minIndex()	*возврат индекса минимального элемента*
int	delMin()	*удаление минимального элемента и возврат его индекса*
boolean	isEmpty()	*пуста ли очередь с приоритетами?*
int	size()	*количество элементов в очереди с приоритетами*

Рис. 2.4.8. *API-интерфейс для обобщенной очереди с приоритетами со связанными индексами*

Такой тип данных удобно реализовать в виде массива, но с быстрым доступом к наименьшему элементу этого массива. Вообще-то он делает даже больше — позволяет быстро выбрать минимальный элемент из *заданного подмножества* элементов массива (вставленных). То есть можно считать, что экземпляр IndexMinPQ по имени pq представляет подмножество массива pq[0..N-1] элементов. Вызов pq.insert(k,item) можно рассматривать как добавление в это подмножество значения k с присваиванием pq[k]=item, а вызов pq.change(k, item) — как присваивание pq[k]=item. Оба вызова используют структуры данных, необходимые для поддержки других операций, из них наиболее важные — delMin() (удаление минимального ключа и возврат его индекса) и change() (изменение элемента, связанного с индексом, который уже находится в

структуре данных — просто как `pq[i] = item`). Эти операции важны во многих приложениях и становятся доступными, поскольку возможно обращение к ключу (по индексу).

В упражнении 2.4.33 описано, как добавить в алгоритм 2.6 реализацию индексной очереди с приоритетами с великолепной эффективностью и небольшим объемом кода. Понятно, что при изменении одного из элементов пирамиды свойство пирамидальности можно восстановить с помощью операции погружения (если ключ увеличился) или всплытия (если ключ уменьшился). Для выполнения этих операций необходимо найти нужный элемент в пирамиде по его индексу. Эта возможность позволяет также добавить в API-интерфейс операцию удаления `delete()`.

Утверждение С (продолжение). Для выполнения операций *вставить*, *изменить приоритет*, *удалить* и *извлечь наименьший* в индексной очереди с приоритетами размером N требуется количество сравнений, пропорциональное не более чем $\log N$.

Доказательство. Непосредственно следует из кода и того факта, что все пути в пирамиде имеют длину не более $\sim\lg N$.

Мы рассмотрели очередь, ориентированную на выявление минимального ключа; на сайте книги приведена и версия `IndexMaxPQ` для выявления максимального ключа.

Таблица 2.4.4. Стоимости операций в худшем случае для индексированной очереди с приоритетами на основе пирамиды из *N* элементов

Операция	Порядок роста для количества сравнений
`insert()`	$\log N$
`change()`	$\log N$
`contains()`	1
`delete()`	$\log N$
`min()`	1
`minIndex()`	1
`delMin()`	$\log N$

Клиент индексной очереди с приоритетами

Клиент `Multiway` класса `IndexMinPQ`, приведенный в листинге 2.4.6, решает задачу *многочастного слияния*: он объединяет несколько отсортированных входных потоков в один упорядоченный выходной поток. Эта задача встречается во многих приложениях, и потоки могут быть замерами, выполненными научными инструментами (упорядоченными по времени), списками информации из Интернета наподобие музыкальных файлов или фильмов (упорядоченными по названиям или именам исполнителя), коммерческими транзакциями (упорядоченными по номерам счетов или времени) и чем угодно еще. При наличии достаточного объема памяти можно просто считать их все в массив и отсортировать, однако очередь с приоритетами позволяет читать входные потоки и объединять их в упорядоченном виде *независимо от их длины*.

Листинг 2.4.6. Клиент очереди с приоритетами для многочастного слияния

```
public class Multiway
{
   public static void merge(In[] streams)
   {
      int N = streams.length;
      IndexMinPQ<String> pq = new IndexMinPQ<String>(N);

      for (int i = 0; i < N; i++)
         if (!streams[i].isEmpty())
             pq.insert(i, streams[i].readString());

      while (!pq.isEmpty())
      {
         StdOut.println(pq.min());
         int i = pq.delMin();
         if (!streams[i].isEmpty())
             pq.insert(i, streams[i].readString());
      }
   }

   public static void main(String[] args)
   {
      int N = args.length;
      In[] streams = new In[N];
      for (int i = 0; i < N; i++)
          streams[i] = new In(args[i]);
      merge(streams);
   }
}
```

Этот клиент класса IndexMinPQ выполняет слияние нескольких отсортированных входных потоков, указанных в аргументах командной строки, в единый упорядоченный выходной поток в стандартном выводе (см. текст). Каждый индекс в потоке связан с ключом (следующая строка в потоке). После инициализации клиент входит в цикл, который выводит строку, наименьшую в очереди, и удаляет соответствующий элемент, а затем добавляет новый элемент для следующей строки в этом потоке. Для экономии места выходные данные показаны (ниже) в одной строке, но реально вывод выполняется построчно.

```
% more m1.txt
A B C F G I I Z
% more m2.txt
B D H P Q Q
% more m3.txt
A B E F J N
```

```
% java Multiway m1.txt m2.txt m3.txt
A A B B B C D E F F G H I I J N P Q Q Z
```

Пирамидальная сортировка

На основе любой очереди с приоритетами можно разработать метод сортировки. Все сортируемые элементы вставляются в очередь с приоритетами, ориентированную на минимум, затем выбираются из нее с помощью повторного применения операции *извлечь наименьший*. Использование очереди с приоритетами, представленной в виде неупорядоченного массива, соответствует сортировке выбором, тогда как применение упорядоченного массива соответствует сортировке вставками. А какой метод сортировки получится при использовании пирамиды? Совершенно другой! Итак, мы применяем пирамидальную структуру данных для разработки классического элегантного алгоритма сортировки, который называется *пирамидальная сортировка* (см. листинг 2.4.7 и рис. 2.4.9).

Листинг 2.4.7. Алгоритм 2.7. Пирамидальная сортировка

```
public static void sort(Comparable[] a)
{
   int N = a.length;
   for (int k = N/2; k >= 1; k--)
      sink(a, k, N);
   while (N > 1)
   {
      exch(a, 1, N--);
      sink(a, 1, N);
   }
}
```

Этот код сортирует элементы `a[1]...a[N]` с помощью метода `sink()` (измененного так, чтобы он принимал в качестве аргументов `a[]` и `N`). Цикл `for` создает пирамиду, а затем цикл `while` обменивает наибольший элемент `a[1]` с `a[N]` и восстанавливает пирамидальность — причем так продолжается до опустошения пирамиды. Уменьшив на единицу индексы массива в реализациях `exch()` и `less()`, можно получить реализацию, которая упорядочивает элементы `a[0]...a[N-1]`, как и все другие сортировки.

Пирамидальная сортировка состоит из двух этапов: *создание пирамиды* (рис. 2.4.10, слева), когда существующий массив преобразуется в пирамиду, и *выстраивание* (рис. 2.4.10, справа), когда элементы извлекаются из пирамиды в порядке убывания, и получается упорядоченный результат. Для согласованности с уже рассмотренным кодом мы будем применять очередь с приоритетами, ориентированную на максимум, и извлекать текущие наибольшие значения одно за другим. Рассматривая эту задачу сортировки, мы временно не будем скрывать представление очереди с приоритетами и будем использовать методы `swim()` и `sink()` непосредственно. Это позволит сортировать массив без задействования дополнительной памяти — можно создавать пирамиду прямо в упорядочиваемом массиве.

Создание пирамиды

Какова сложность процесса построения пирамиды из N заданных элементов? Очевидно, это можно сделать за время, пропорциональное $N\log N$: нужно перебирать элементы массива слева направо и обеспечивать с помощью метода `swim()`, чтобы элементы слева от указателя перебора всегда составляли пирамидально упорядоченное полное дерево — вроде последовательных вставок в очередь с приоритетами.

		a[i]											
N	k	0	1	2	3	4	5	6	7	8	9	10	11
Первоначальные значения			S	O	R	T	E	X	A	M	P	L	E
11	5		S	O	R	T	L	X	A	M	P	E	E
11	4		S	O	R	T	L	X	A	M	P	E	E
11	3		S	O	X	T	L	R	A	M	P	E	E
11	2		S	T	X	P	L	R	A	M	O	E	E
11	1		X	T	S	P	L	R	A	M	O	E	E
Пирамидальная упорядоченность			X	T	S	P	L	R	A	M	O	E	E
10	1		T	P	S	O	L	R	A	M	E	E	X
9	1		S	P	R	O	L	E	A	M	E	T	X
8	1		R	P	E	O	L	E	A	M	S	T	X
7	1		P	O	E	M	L	E	A	R	S	T	X
6	1		O	M	E	A	L	E	P	R	S	T	X
5	1		M	L	E	A	E	O	P	R	S	T	X
4	1		L	E	E	A	M	O	P	R	S	T	X
3	1		E	A	E	L	M	O	P	R	S	T	X
2	1		E	A	E	L	M	O	P	R	S	T	X
1	1		A	E	E	L	M	O	P	R	S	T	X
Отсортированный результат			A	E	E	L	M	O	P	R	S	T	X

Рис. 2.4.9. *Трассировка пирамидальной сортировки (содержимое массива после каждого погружения)*

Но существует более хитроумный и гораздо более эффективный способ: выполнять перебор справа налево и вызывать метод `sink()` для создания пирамидальных подструктур. Каждую позицию в массиве можно считать корнем маленькой пирамиды, а метод `sink()` работает и для таких пирамид. Если два потомка узла являются пирамидами, то вызов `sink()` для этого узла создает поддерево с корнем в родительском узле. Этот процесс индуктивно наводит пирамидальный порядок. Обратный перебор можно начать с середины массива, чтобы пропустить подпирамиды размером 1, и закончить в позиции 1, после завершения построения пирамиды одним вызовом `sink()`. Такая сборка пирамиды выглядит несколько неестественно, т.к. ее цель состоит в получении пирамидальной упорядоченности, при которой наибольший элемент находится в первой позиции массива (а другие большие элементы тоже близко к началу), а не в его конце.

Утверждение Т. Создание пирамиды из N элементов с помощью погружений требует менее $2N$ сравнений и менее N перестановок.

Доказательство. Это следует из наблюдения, что большинство обрабатываемых пирамид имеют малый размер. Например, для построения пирамиды из 127 элементов необходимо обработать 32 пирамиды размером 3, 16 пирамид размером 7, 8 пирамид размером 15, 4 пирамиды размером 31, 2 пирамиды размером 63 и 1 пирамиду размером 127. Поэтому необходимо выполнить

$32 \cdot 1 + 16 \cdot 2 + 8 \cdot 3 + 4 \cdot 4 + 2 \cdot 5 + 1 \cdot 6 = 120$ перестановок

(и вдвое больше сравнений) в худшем случае. Полному доказательству посвящено упражнение 2.4.20.

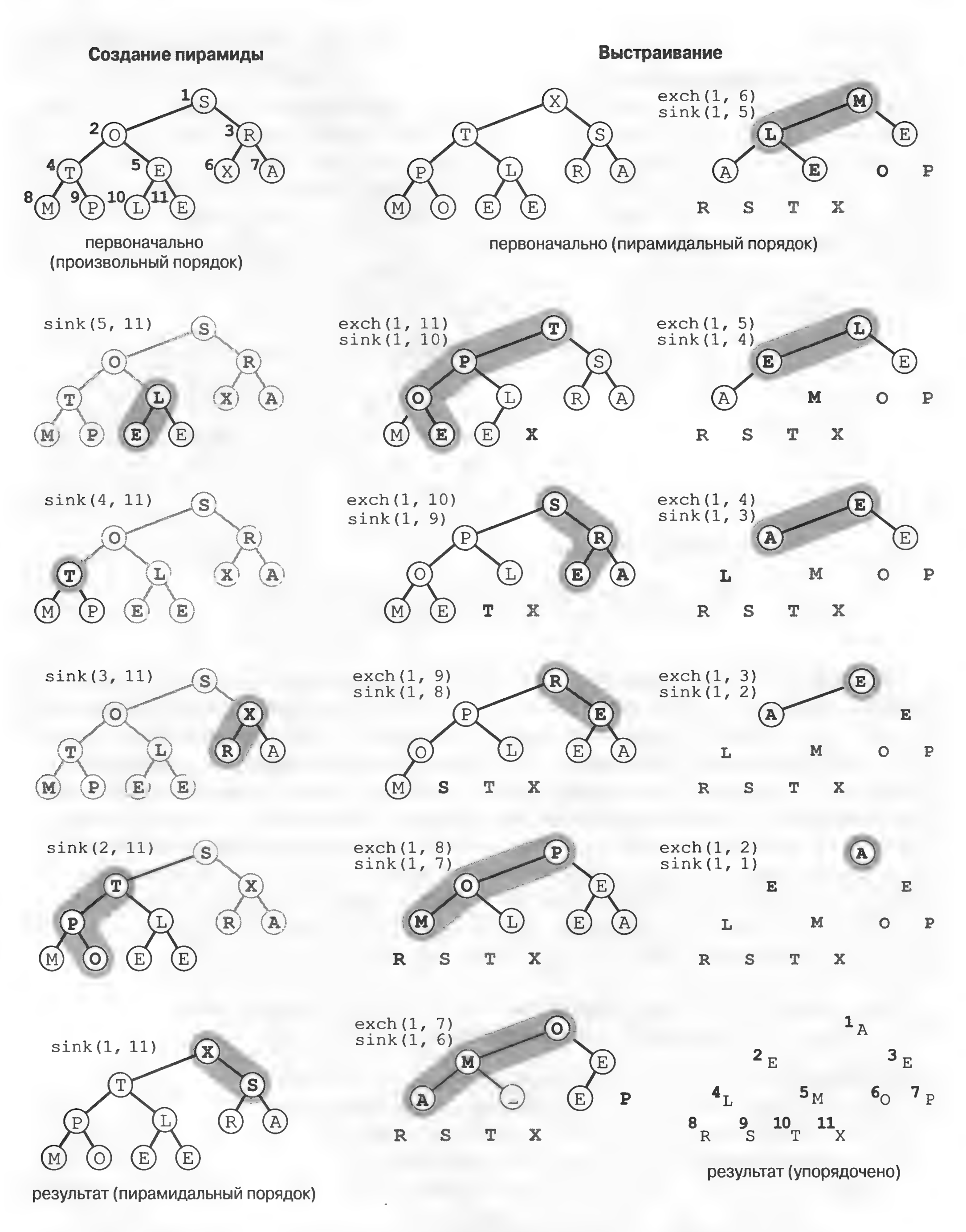


Рис. 2.4.10. Пирамидальная сортировка: создание пирамиды (слева) и выстраивание (справа)

Выстраивание

Основная часть работы при пирамидальной сортировке выполняется на втором этапе, когда из пирамиды изымается наибольший оставшийся элемент и помещается в элемент массива, освободившийся в результате сжатия пирамиды. Этот процесс немного похож на сортировку выбором (элементы выбираются не по возрастанию, а по убыванию), но в нем происходит значительно меньше сравнений, т.к. пирамида гораздо эффективнее находит наибольший элемент в неупорядоченной части массива.

Утверждение У. Для упорядочения N элементов пирамидальная сортировка выполняет менее $2N\lg N + 2N$ сравнений (и вдвое меньше перестановок).

Доказательство. Слагаемое $2N$ отражает стоимость создания пирамиды (см. утверждение Т). Слагаемое $2N\lg N$ следует из ограничения стоимости каждой операции погружения при выстраивании величиной $2\lg N$ (см. утверждение С).

Алгоритм 2.7 представляет собой полную реализацию на основе всех изложенных соображений — это классический алгоритм *пирамидальной сортировки*, который был разработан Дж. Вильямсом (J. W. J. Williams) и усовершенствован Р. Флойдом (R. W. Floyd) в 1964 г. Циклы в этой программе выполняют с виду разные действия (первый создает пирамиду, а второй выстраивает данные, разрушая пирамиду), но они оба построены на основе метода `sink()`. Реализация вынесена за пределы нашего API-интерфейса для очереди с приоритетами, чтобы подчеркнуть простоту алгоритма сортировки: восемь строк кода для метода `sort()` и еще восемь — для метода `sink()` выполняют сортировку массива на месте.

Как обычно, нагляднее понять действие алгоритма помогает визуальная трассировка (рис. 2.4.11). Вначале процесс выполняет какие-то действия, совершенно не похожие на сортировку, т.к. при создании пирамиды бо́льшие элементы сдвигаются к началу массива. Но потом метод становится похож на зеркальное отражение сортировки выбором (только использует гораздо меньше сравнений).

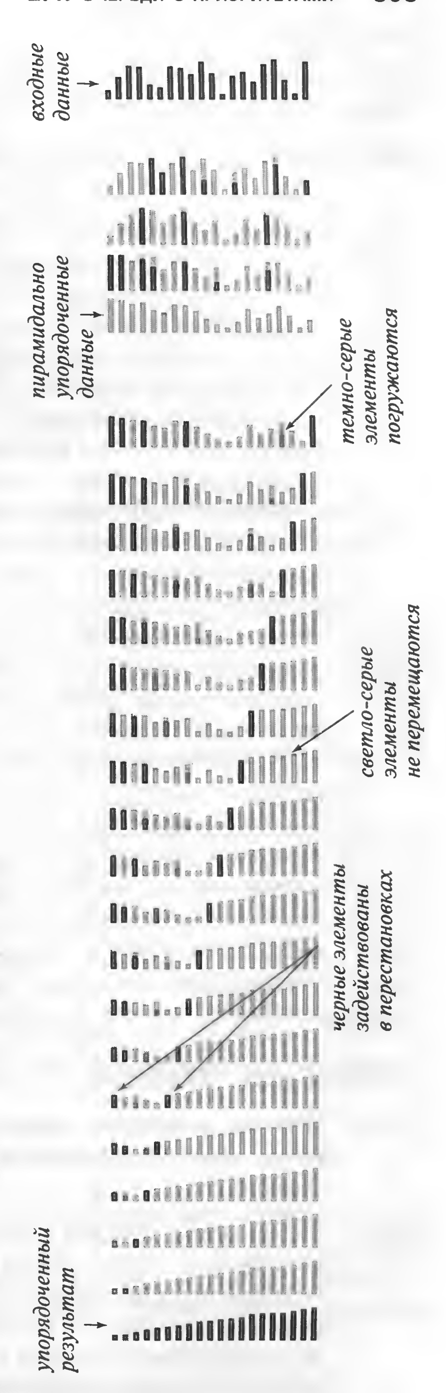


Рис. 2.4.11. Визуальная трассировка пирамидальной сортировки

Как и в случае всех других рассмотренных нами методов, многие люди искали способы усовершенствовать реализации очереди с приоритетами и пирамидальной сортировки. Сейчас мы кратко рассмотрим один из них.

Погружение до основания, а затем всплытие

Большинство элементов, снова вставляемых в пирамиду во время выстраивания, погружаются до самого основания. В 1964 г. Флойд заметил, что можно сэкономить время на проверке, достиг ли элемент своей позиции, просто повышая больший из двух потомков до самого основания, а затем возвращаясь вверх до нужной позиции. Эта идея сокращает количество сравнений асимптотически вдвое — близко к количеству, используемому сортировкой слиянием (для случайно упорядоченного массива). Такой метод требует некоторых дополнительных действий и на практике удобен только в случае относительно трудоемких сравнений (например, при сортировке элементов со строками или другими видами длинных ключей).

Пирамидальная сортировка важна при изучении сложности сортировки (см. подраздел "Сложность сортировки" в разделе 2.2), поскольку это единственный метод из рассмотренных в данной книге, который оптимален (с некоторым постоянным коэффициентом) относительно как времени, так и памяти: она гарантированно выполняет $\sim 2N\lg N$ сравнений и использует постоянный объем дополнительной памяти даже в худшем случае. Пирамидальная сортировка популярна при очень ограниченной памяти (например, во встроенной системе или в дешевом смартфоне), т.к. ее можно реализовать каким-то десятком строк (даже в машинном коде), и все-таки получить максимальную производительность. Однако в типичных приложениях в современных системах она используется нечасто из-за ухудшения производительности кеша: элементы массива редко сравниваются с соседними элементами, поэтому количество промахов кеша гораздо больше, чем для быстрой сортировки, сортировки слиянием и даже сортировки Шелла, где большинство сравнений выполняются с соседними элементами.

А вот использование пирамидальных структур для реализации очередей с приоритетами играет все более важную роль в современных приложениях, поскольку они позволяют легко гарантировать логарифмическое время выполнения в динамических ситуациях с непредсказуемой смесью операций *вставить* и *извлечь наибольший*. Несколько подобных примеров мы еще рассмотрим в данной книге.

Вопросы и ответы

Вопрос. **Мне так до конца и непонятно назначение очередей с приоритетами. Почему нельзя просто отсортировать элементы и затем перебирать их в упорядоченном массиве?**

Ответ. В некоторых случаях обработки данных — таких как программы `TopM` и `Multiway` — общий объем данных неподъемно велик не только для их сортировки, но даже и просто для хранения в памяти. Если нужно найти десять наибольших элементов из миллиардов, то стоит ли сортировать все миллиарды? Ведь это же можно сделать с очереди с приоритетами из десяти элементов. Бывает и так, что все данные даже не существуют одновременно: мы выбираем несколько элементов из очереди с приоритетами, обрабатываем их, и результаты обработки, возможно, снова добавляются в эту очередь.

Вопрос. **Почему в классе `MaxPQ` используется обобщенный тип `Item`, а не просто `Comparable`?**

Ответ. Это потребовало бы выполнения в клиенте приведения типа, возвращаемого функцией `delMax()`, в реальный тип, подобный `String`. В общем случае следует избегать приведения типов в клиентском коде.

Вопрос. **Почему в представлении пирамиды не используется элемент `a[0]`?**

Ответ. Это слегка упрощает арифметику. Не очень трудно реализовать пирамидальные методы с отсчетом элементов с нуля, когда потомками `a[0]` являются `a[1]` и `a[2]`, потомками `a[1]` — элементы `a[3]` и `a[4]`, потомками `a[2]` — `a[5]` и `a[6]` и т.д. Но большинство программистов, как и мы, предпочитают использовать более простые арифметические выражения. А в некоторых применениях пирамидальных структур элемент `a[0]` удобно использовать в качестве сигнального элемента (как родителя `a[1]`).

Вопрос. **Мне кажется, что создание пирамиды в пирамидальной сортировке проще выполнять поочередной вставкой элементов, чем запутанным восходящим методом. Зачем он нужен?**

Ответ. В реализациях сортировки восходящий метод выполняется на 20% быстрее и требует вдвое меньше хитроумного кода (не нужен метод `swim()`). Сложность понимания алгоритма совсем не означает, что сам алгоритм сложен или неэффективен.

Вопрос. **Что произойдет, если в реализации наподобие `MaxPQ` убрать фразу `extends Comparable<Key>`?**

Ответ. Как обычно, ответ на этот вопрос проще всего получить практически. Если сделать так, вы получите ошибку компиляции:

```
MaxPQ.java:21: cannot find symbol
symbol  : method compareTo(Item)
```

MaxPQ.java:21: не удается найти символ
символ : метод compareTo(Item)

Это означает, что компилятор Java не знает метод `compareTo()` для класса `Item` — ведь вы не указали в объявлении, что он реализует интерфейс `Comparable<Item>`.

Упражнения

2.4.1. К первоначально пустой очереди с приоритетами применяется последовательность операций P R I O * R * * I * T * Y * * * Q U E * * * U * E, где буква означает операцию *вставить* (эту букву), а звездочка — операцию *извлечь наибольший*. Приведите последовательность букв, которые будут возвращены операциями *извлечь наибольший*.

2.4.2. Оцените эффективность следующей идеи. Для реализации операции *найти наибольший* с постоянным временем выполнения можно использовать стек или очередь, но отслеживать максимальное из вставленных туда значений, а затем, при выполнении операции *найти наибольший*, возвратить это значение.

2.4.3. Приведите реализации очередей с приоритетами, которая поддерживает операции *вставить* и *извлечь наибольший*, для каждой из следующих базовых структур данных: неупорядоченный массив, упорядоченный массив, неупорядоченный связный список и упорядоченный связный список. Приведите таблицу с граничными значениями времени выполнения для каждой операции и для каждой из четырех реализаций.

2.4.4. Является ли массив, упорядоченный по убыванию, пирамидой, ориентированной на максимум?

2.4.5. Приведите содержимое пирамиды, которая получится после вставки ключей `E A S Y Q U E S T I O N` в этом порядке в первоначально пустую пирамиду, ориентированную на максимум.

2.4.6. Приведите последовательность пирамид, которые получатся при выполнении операций `P R I O * R * * I * T * Y * * * Q U E * * * U * E` в первоначально пустой пирамиде, ориентированной на максимум (обозначения те же, что и в упражнении 2.4.1).

2.4.7. Наибольший элемент пирамиды должен находиться в позиции 1, а второй — в позиции 2 или 3. Приведите список позиций в пирамиде размером 31, где k-й наибольший элемент (1) может появиться и (2) не может появиться, для $k = 2$, 3, 4 (все значения в пирамиде различны).

2.4.8. Ответьте на вопросы предыдущего упражнения для k-го *наименьшего* элемента.

2.4.9. Нарисуйте все различные пирамиды, которые можно собрать из пяти ключей `A B C D E`, а затем — все различные пирамиды, которые можно собрать из пяти ключей `A A A B B`.

2.4.10. Допустим, мы не хотим, чтобы первая позиция в пирамидально упорядоченном массиве `pq[]` оставалась не при деле, и будем помещать наибольшее значение в `pq[0]`, его потомков — в `pq[1]` и `pq[2]`, и т.д. по уровням. Где находятся родитель и потомки элемента `pq[k]`?

2.4.11. Некоторое приложение выполняет очень много операций *вставить*, но лишь несколько операций *извлечь наибольший*. Какая реализация очереди с приоритетами будет в этом случае наиболее эффективна — пирамида, неупорядоченный массив или упорядоченный массив?

2.4.12. Некоторое приложение выполняет очень много операций *найти наибольший*, но относительно немного операций *вставить* и *извлечь наибольший*. Какая реализация очереди с приоритетами будет в этом случае наиболее эффективна — пирамида, неупорядоченный массив или упорядоченный массив?

2.4.13. Как можно избежать проверки `j<N` в методе `sink()`?

2.4.14. Какое минимальное количество элементов необходимо обменять местами при выполнении операции *извлечь наибольший* в пирамиде размером N без повторяющихся ключей? Приведите пирамиду размером 15, для которой достигается это минимальное количество. Ответьте на эти же вопросы для двух и трех последовательных операций *извлечь наибольший*.

2.4.15. Придумайте алгоритм с линейным временем выполнения, который проверяет, является ли массив `pq[]` пирамидой, ориентированной на минимум.

2.4.16. Приведите массивы из $N = 32$ элементов, для которых пирамидальная сортировка выполняет *максимальное* и *минимальное* количество сравнений.

2.4.17. Докажите, что создание очереди с приоритетами, ориентированной на минимум, размером k, а затем выполнение $N-k$ операций *заменить наименьший* (*вставить* с последующей операцией *извлечь наименьший*) оставляет в очереди с приоритетами k наибольших из N элементов.

2.4.18. Пусть клиент класса `MaxPQ` вызывает метод `insert()` с элементом, который больше всех элементов в очереди без повторяющихся элементов, а затем сразу же вызывает метод `delMax()`. Будет ли полученная пирамида идентична первоначальной пирамиде? Ответьте на тот же вопрос для случая двух операций `insert()` (первая вставляет ключ, больший всех ключей в очереди, а вторая вставляет еще больший ключ), после которых выполняются две операции `delMax()`.

2.4.19. Реализуйте конструктор для класса `MaxPQ`, который принимает в качестве аргумента массив элементов и использует метод восходящей сборки пирамиды, описанный в тексте.

2.4.20. Докажите, что создание пирамиды с помощью операции погружения выполняет менее $2N$ сравнений и менее N перестановок.

Творческие задачи

2.4.21. *Элементарные структуры данных.* Объясните, как можно использовать очередь с приоритетами для реализации типов данных стека, очереди и рандомизованной очереди (см. главу 1).

2.4.22. *Изменение размера массива.* Добавьте в класс `MaxPQ` изменение размера массива и обоснуйте граничные значения, как в утверждении С, для количества обращений к массиву (в амортизированном смысле).

2.4.23. *Многочастные пирамиды.* Учитывая только стоимость сравнений и предполагая, что для нахождения наибольшего из t элементов требуется t сравнений, найдите значение t, которое минимизирует коэффициент перед $N\lg N$ в счетчике сравнений при использовании в пирамидальной сортировке t-арной пирамиды. Сначала рассмотрите естественное обобщение метода `sink()`, а затем считайте, что метод Флойда может сэкономить одно сравнение во внутреннем цикле.

2.4.24. *Явные ссылки в очереди с приоритетами.* Реализуйте очередь с приоритетами с помощью пирамидально упорядоченного бинарного дерева, но используйте не массив, а структуру с тройными связями. Каждый узел должен содержать три ссылки: две для переходов вниз по дереву и одну для перехода вверх. Ваша реализация должна гарантировать логарифмическое время выполнения одной операции, даже если максимальный размер очереди с приоритетами заранее не известен.

2.4.25. *Вычисления в теории чисел.* Напишите программу `CubeSum.java`, которая выводит все целые числа вида $a^3 + b^3$ (где a и b — целые числа от 0 до N), упорядоченные по возрастанию, без использования большого объема вспомогательной памяти. То есть нужно не вычислять массив N^2 сумм и потом сортировать его,

а создать очередь с приоритетами, ориентированную на минимум, которая вначале содержит элементы $(0^3, 0, 0)$, $(1^3, 1, 0)$, $(2^3, 2, 0)$, ..., $(N^3, N, 0)$. Затем, пока очередь не пуста, *извлеките наименьший* элемент $(i^3 + j^3, i, j)$, выведите его и, если $j < N$, *вставьте* элемент $(i^3 + (j + 1)^3, i, j + 1)$. Используйте эту программу, чтобы найти все различные целочисленные a, b, c и d от 0 до 10^6, такие, что $a^3 + b^3 = c^3 + d^3$.

2.4.26. *Пирамида без перестановок.* Примитив `exch()` используется в операциях `sink()` и `swim()`, и поэтому элементы загружаются и сохраняются в два раза чаще, чем это необходимо. Приведите более эффективные реализации, без этого недостатка, как в сортировке вставками (см. упражнение 2.1.25).

2.4.27. *Поиск минимума.* Добавьте в класс `MaxPQ` метод `min()`. Он должен использовать постоянное время и постоянный объем дополнительной памяти.

2.4.28. *Фильтр точек.* Напишите клиент класса `TopM`, который считывает из стандартного ввода точки (x, y, z), принимает из командной строки значение M и выводит M точек, которые находятся ближе всего к началу координат (евклидово расстояние). Оцените время выполнения этого клиента для $N = 10^8$ и $M = 10^4$.

2.4.29. *Минимаксная очередь с приоритетами.* Разработайте тип данных, который поддерживает следующие операции: *вставить, удалить наибольший* и *удалить наименьший* (все за логарифмическое время) и *найти наибольший* и *найти наименьший* (обе за постоянное время). *Совет*: используйте две пирамиды.

2.4.30. *Динамическое определение медианы.* Разработайте тип данных, который поддерживает операцию *вставить* за логарифмическое время, *найти медиану* за постоянное время и *удалить медиану* за логарифмическое время. *Совет*: используйте пирамиду, ориентированную на минимум, и пирамиду, ориентированную на максимум.

2.4.31. *Быстрая вставка.* Разработайте для API-интерфейса `MinPQ` реализацию на основе сравнений, такую, чтобы операция *вставить* выполняла $\sim\log \log N$ сравнений, а операция *удалить наименьший* выполняла $\sim 2\log N$ сравнений. *Совет*: для поиска предка в методе `swim()` используйте бинарный поиск по родительским указателям.

2.4.32. *Нижняя граница.* Докажите, что невозможно разработать реализацию на основе для API-интерфейса `MinPQ` сравнений, в которой обе операции *вставить* и *удалить наименьший* гарантированно выполняли бы $\sim N \log \log N$ сравнений.

2.4.33. *Реализация индексной очереди с приоритетами.* Реализуйте базовые операции из API-интерфейса индексной очереди с приоритетами (см. рис. 2.4.8), изменив алгоритм 2.6 следующим образом. Пусть массив `pq[]` хранит индексы, а ключи хранятся в дополнительном массиве `keys[]`. И добавьте еще массив `qp[]`, обратный массиву `pq[]`: `qp[i]` содержит позицию для `i` из `pq[]`, т.е. индекс `j`, такой, что `pq[j]` содержит `i`). Затем добавьте в алгоритм 2.6 код, работающий с этими структурами данных. Используйте соглашение, что `qp[i] = -1`, если `i` не находится в очереди, и добавьте метод `contains()`, который проверяет это условие. При этом понадобится изменить вспомогательные методы `exch()` и `less()`, но не `sink()` и `swim()`.

Частичное решение

```
public class IndexMinPQ<Key extends Comparable<Key>>
{
   private int N;               // количество элементов в ОсП
   private int[] pq;            // бинарная пирамида с индексацией с 1
   private int[] qp;            // обращение: qp[pq[i]] = pq[qp[i]] = i
   private Key[] keys;          // элементы с приоритетами

   public IndexMinPQ(int maxN)
   {
      keys = (Key[]) new Comparable[maxN + 1];
      pq   = new int[maxN + 1];
      qp   = new int[maxN + 1];
      for (int i = 0; i <= maxN; i++) qp[i] = -1;
   }

   public boolean isEmpty()
   {  return N == 0;  }

   public boolean contains(int k)
   {  return qp[k] != -1;  }

   public void insert(int k, Key key)
   {
      N++;
      qp[k] = N;
      pq[N] = k;
      keys[k] = key;
      swim(N);
   }

   public Item min()
   {  return keys[pq[1]];  }

   public int delMin()
   {
      int indexOfMin = pq[1];
      exch(1, N--);
      sink(1);
      keys[pq[N+1]] = null;
      qp[pq[N+1]] = -1;
      return indexOfMin;
   }
}
```

2.4.34. *Реализация индексной очереди с приоритетами (дополнительные операции).* Добавьте в реализацию из упражнения 2.4.33 операции `minIndex()`, `change()` и `delete()`.

Решение

```
public int minIndex()
{  return pq[1];  }

public void change(int k, Item item)
{
   keys[k] = key;
   swim(qp[k]);
   sink(qp[k]);
}
```

```
public void delete(int k)
{
   exch(k, N--);
   swim(qp[k]);
   sink(qp[k]);
   keys[pq[N+1]] = null;
   qp[pq[N+1]] = -1;
}
```

2.4.35. *Выборка из дискретного распределения вероятности.* Напишите класс `Sample` с конструктором, который принимает в качестве аргумента массив `p[]` значений `double` и поддерживает следующие две операции: `random()`, возвращающую индекс `i` с вероятностью `p[i]/T` (где `T` — сумма чисел в `p[]`), и `change(i, v)`, заменяющую значение `p[i]` на `v`. *Совет*: используйте полное бинарное дерево, где каждому узлу приписан вес `p[i]`. Храните в каждом узле сумму весов всех узлов в его поддереве. Чтобы сгенерировать случайный индекс, выберите случайное число от 0 до `T` и рассматривайте ветви поддерева на основе их суммарных весов. При изменении `p[i]` необходимо изменить все веса узлов на пути от корня до этого `i`. Не используйте явных указателей, как это делается в пирамидах.

Эксперименты

2.4.36. *Драйвер производительности I.* Напишите клиентскую программу — драйвер производительности, которая заполняет очередь с приоритетами с помощью операций *вставить*, потом удаляет половину ключей с помощью операций *извлечь наибольший*, затем снова заполняет очередь операциями *вставить* и удаляет все ключи операциями *извлечь наибольший* — и так много раз для случайных последовательностей ключей различной длины от коротких до длинных. Программа должна замерять время каждого выполнения и выводить (численно или графиком) средние значения времени выполнения.

2.4.37. *Драйвер производительности II.* Напишите клиентскую программу — драйвер производительности, которая заполняет очередь с приоритетами с помощью операций *вставить*, потом выполняет столько операций *извлечь наибольший* и *вставить*, сколько возможно за 1 секунду — и так много раз для случайных последовательностей ключей различной длины от коротких до длинных. Программа должна выводить (численно или графиком) средние количества операций *извлечь наибольший*, которые она смогла выполнить.

2.4.38. *Драйвер экстремальных проверок.* Напишите клиентскую программу — драйвер проверок, которая использует интерфейс очереди с приоритетами из алгоритма 2.6 для сложных или патологических случаев, которые могут возникнуть в реальных ситуациях. Вот несколько примеров: уже упорядоченные ключи, ключи в обратном порядке, все одинаковые ключи и последовательности ключей из только двух различных значений.

2.4.39. *Стоимость создания.* Эмпирически определите процент времени, которое пирамидальная сортировка тратит на этап создания пирамиды, для $N = 10^3$, 10^6 и 10^9.

2.4.40. *Метод Флойда.* Реализуйте версию пирамидальной сортировки на основе предложенной Флойдом идеи погружения до основания, а затем всплытия. Подсчитайте количество сравнений, которое выполнила программа, и количество сравнений, использованное стандартной реализацией, для случайно упорядоченных различных ключей, для $N = 10^3$, 10^6 и 10^9.

2.4.41. *Многочастные пирамиды.* Реализуйте версию пирамидальной сортировки на основе полных пирамидально упорядоченных 3-арных и 4-арных деревьев (см. в тексте раздела). Подсчитайте количество сравнений, которое выполнил каждый вариант, и количество сравнений, использованное стандартной реализацией, для случайно упорядоченных различных ключей, для $N = 10^3$, 10^6 и 10^9.

2.4.42. *Пирамиды с прямым обходом.* Реализуйте версию пирамидальной сортировки, в которой пирамидально упорядоченное дерево находится в прямом порядке, а не по уровням. Подсчитайте количество сравнений, которое выполнила программа, и количество сравнений, использованное стандартной реализацией, для случайно упорядоченных различных ключей, для $N = 10^3$, 10^6 и 10^9.

2.5. Применения

Алгоритмы сортировки и очереди с приоритетами широко применяются в самых различных приложениях. В этом разделе мы кратко рассмотрим некоторые такие приложения, узнаем, какую важную роль играют рассмотренные нами методы в этих приложениях, и обсудим шаги, необходимые для использования нашего кода методов сортировки и очередей с приоритетами.

Основная причина полезности сортировки в том, что элементы гораздо легче искать в упорядоченном массиве, чем в произвольном. Люди давно поняли, что проще искать номер чьего-то телефона, если строки в нем упорядочены по фамилиям. Теперь цифровые музыкальные плееры упорядочивают музыкальные файлы по фамилиям исполнителей или по названиям композиций, поисковые механизмы выводят результаты поиска по убыванию релевантности, электронные таблицы могут упорядочивать данные по конкретным полям, пакеты матричных вычислений сортируют вещественные собственные значения симметричных матриц по убыванию и т.д. Есть и другие задачи, решение которых облегчается, если массив упорядочен: поиск термина в предметном указателе книги, удаление повторяющихся элементов в длинном списке вроде списка рассылки, списка избирателей или списка веб-сайтов, или выполнение статистических расчетов вроде удаления выбросов, поиск медианы или вычисление процентиля.

Сортировка также встречается в виде критического компонента во многих приложениях, которые вроде бы вообще никак не связаны с сортировкой. Вот лишь несколько примеров: сжатие данных, компьютерная графика, вычислительная биология, управление поставками, комбинаторная оптимизация, выборы и голосование. Алгоритмы, с которыми вы познакомились в данной главе, играют важную роль в разработке эффективных алгоритмов в каждой из последующих глав книги.

Наиболее важна системная сортировка, поэтому мы начнем с рассмотрения ряда практических соображений, которые следует учитывать при создании сортировки, если ее предполагается использовать в широком спектре клиентов. Некоторые из этих тем характерны для Java, но они отражают трудности, которые придется преодолевать в любой системе.

Главное, что мы хотим продемонстрировать — что, несмотря на использование относительно простых механизмов, изучаемые нами реализации широко применимы. Список естественных приложений быстрых сортирующих алгоритмов весьма обширен, и мы можем рассмотреть лишь небольшую часть из них — немного научных, немного алгоритмических и немного коммерческих. Еще много дополнительных примеров содержат упражнения, и еще больше — сайт книги. Более того, часто даже *в данной книге* мы будем ссылаться на эту главу, чтобы найти эффективное решение задачи!

Сортировка различных видов данных

Наши реализации сортируют массивы объектов `Comparable`. Это соглашение, принятое в Java, позволяет использовать механизм *обратных вызовов* Java для сортировки массивов объектов любого типа, который реализует интерфейс `Comparable`. Как сказано в разделе 2.1, реализация этого интерфейса сводится к определению метода `compareTo()`, который реализует *естественное упорядочение* для этого типа. Наш

код может непосредственно упорядочивать массивы типов String, Integer, Double и любых других типов вроде File и URL, т.к. они реализуют интерфейс Comparable. Возможность использовать один и тот же код для всех типов — удобная вещь, но обычно приложения работают с типами данных, которые определяются внутри этих приложений. Поэтому программисты часто добавляют для своих данных метод compareTo(), чтобы реализовать интерфейс Comparable, и тогда клиентский код может сортировать массивы этого типа (и создавать очереди с приоритетами из значений этого типа).

Пример с транзакциями

Характерным местом обитания сортирующих приложений является обработка коммерческих данных. К примеру, компания, занимающаяся электронной коммерцией, может хранить для каждой транзакции запись, в которой указана учетная запись клиента, содержащая всю остальную нужную информацию — имя клиента, дата, сумма и т.д. В наши дни успешной компании может понадобиться обрабатывать многие миллионы таких транзакций. Как было сказано в упражнении 2.1.21, разумно определить естественный порядок таких транзакций по суммам — это можно реализовать с помощью добавления соответствующего метода compareTo() в определение класса. После этого можно обработать массив a[] объектов Transaction, к примеру, сначала отсортировав его с помощью вызова Quick.sort(a). Наши методы сортировки ничего не знают о типе данных Transaction, но интерфейс Comparable позволяет определить естественный порядок, и тогда для упорядочения объектов Transaction можно использовать любой из наших методов. Либо можно указать, что объекты Transaction следует упорядочивать по дате, для этого в методе compareTo() нужно сравнивать поля Date. Объекты Date уже поддерживают интерфейс Comparable, поэтому можно вызывать готовый метод compareTo(), а не реализовать его с нуля (листинг 2.5.1). Можно также выполнять сортировку данных по полю клиента — так можно предоставить клиентам возможность переключаться между несколькими различными заказами, но тут возникают интересные моменты, которые мы вскоре рассмотрим.

Листинг 2.5.1. Альтернативная реализация метода compareTo() для сортировки транзакций по датам

```
public int compareTo(Transaction that)
{  return this.when.compareTo(that.when);  }
```

Сортировка указателей

Применяемый нами подход называется в классической литературе *сортировкой указателей* — потому что обрабатываются ссылки на элементы, а сами данные остаются на своих местах. В языках программирования наподобие С и С++ программисты сами решают, с чем им работать — с данными или указателями на них. В Java по умолчанию обрабатываются указатели. Во всех случаях, кроме примитивных числовых типов, мы *всегда* работаем со ссылками на объекты (указателями), а не с самими объектами. Сортировка указателей добавляет промежуточный уровень, и массивы содержат ссылки на сортируемые объекты, а не сами объекты. Мы кратко рассмотрим некоторые связанные с этим моменты, на которые следует обратить внимание при сортировке. Имея несколько массивов ссылок, можно получить несколько по-разному упорядоченных представлений одного и того же набора данных или его отдельных частей (возможно, используя различные ключи, как сказано ниже).

Ключи неизменны

Понятно, что упорядоченность массива может быть нарушена, если клиенту разрешено изменять значения ключей после сортировки. Точно так же и очередь с приоритетами вряд ли будет правильно работать, если клиент сможет изменять значения ключей между операциями. В Java удобно использовать неизменные ключи. Большинство стандартных типов данных, которые обычно применяются в качестве ключей — наподобие `String`, `Integer`, `Double` и `File` — как раз неизменны.

Перестановки имеют низкую стоимость

Еще одно преимущество использования ссылок — при этом не надо перемещать сами элементы. Экономия может быть весьма существенной для массивов с большими элементами (и небольшими ключами), т.к. для сравнения необходим доступ лишь к небольшой части элемента, а все остальное никак не затрагивается во время сортировки. Способ со ссылками делает стоимость обмена примерно равной стоимости сравнения в обычных случаях, даже при произвольно больших ключах (за счет дополнительной памяти, необходимой для ссылок). А при длинных ключах обмены могут оказаться даже менее трудоемкими, чем сравнения. Один из способов оценить производительность алгоритмов, которые сортируют массивы чисел — просто узнать общее количество сравнений и обменов, которые они выполняют, и принять, что стоимость обменов совпадает со стоимостью сравнений. Сделанные таким образом выводы обычно применимы к широкому классу Java-приложений, поскольку в них выполняется сортировка ссылок на объекты.

Альтернативные упорядочения

Существует много приложений, в которых в зависимости от ситуации требуются различные упорядочения объектов. Интерфейс `Comparator` позволяет создавать для одного класса несколько упорядочений. В нем имеется единственный общедоступный метод `compare()`, который выполняет сравнение двух объектов. Если мы работаем с типом данных, реализующим этот интерфейс, то его можно передавать в метод `sort()` (который передает его в метод `less()`), как в листинге 2.5.2. Механизм `Comparator` позволяет сортировать массивы объектов произвольного типа на основе любого полного порядка, который мы определим для этого типа. Использование интерфейса `Comparator` вместо работы с типами `Comparable` полнее отделяет определение типа от определения операции сравнения двух объектов этого типа. А поскольку может существовать много разных способов сравнения объектов, механизм `Comparator` позволяет выбрать любой из них. Например, для упорядочения массива `a[]` строк без учета регистра букв можно просто записать вызов `Insertion.sort(a, String.CASE_INSENSITIVE_ORDER)`, в котором используется компаратор `CASE_INSENSITIVE_ORDER`, определенный в Java-классе `String`.

Листинг 2.5.2. Сортировка вставками с помощью компаратора

```
public static void sort(Object[] a, Comparator c)
{
   int N = a.length;
   for (int i = 1; i < N; i++)
      for (int j = i; j > 0 && less(c, a[j], a[j-1]); j--)
         exch(a, j, j-1);
}
```

```
private static boolean less(Comparator c, Object v, Object w)
{  return c.compare  (v, w) < 0;  }

private static void exch(Object[] a, int i, int j)
{  Object t = a[i]; a[i] = a[j]; a[j] = t; }
```

Элементы с несколькими ключами

В типичных приложениях элементы содержат несколько переменных экземпляров, которые могут служить в качестве ключей сортировки. В нашем примере с транзакциями одному клиенту может понадобиться упорядочить список транзакций по фамилиям (например, чтобы собрать вместе все транзакции для каждого покупателя), другому — по суммам (например, чтобы сразу увидеть самые крупные продажи), а третьему — по каким-то другим полям. Для обеспечения этой гибкости идеально подходит механизм `Comparator`. Можно определить несколько компараторов, как в альтернативной реализации класса `Transaction`, приведенной в листинге 2.5.3. При таком определении клиент может отсортировать массив объектов `Transaction` по времени с помощью вызова

```
Insertion.sort(a, new Transaction.WhenOrder())
```

или по сумме с помощью вызова

```
Insertion.sort(a, new Transaction.HowMuchOrder())
```

Листинг 2.5.3. Набор компараторов для сортировки вставками

```
import java.util.Comparator;
public class Transaction
{
   ...
   private final String who;
   private final Date when;
   private final double amount;
   ...
   public static class WhoOrder implements Comparator<Transaction>
   {
      public int compare(Transaction v, Transaction w)
      {  return v.who.compareTo(w.when);  }
   }

   public static class WhenOrder implements Comparator<Transaction>
   {
      public int compare(Transaction v, Transaction w)
      {  return v.when.compareTo(w.when);  }
   }

   public static class HowMuchOrder implements Comparator<Transaction>
   {
      public int compare(Transaction v, Transaction w)
      {
         if (v.amount < w.amount) return -1;
         if (v.amount > w.amount) return +1;
         return 0;
      }
   }
}
```

Сортировка выполняет упорядочивание с помощью *обратного вызова* метода `compare()` из класса `Transaction`, который указан в клиентском коде. Чтобы не создавать новый объект `Comparator` для каждой сортировки, для определения компараторов можно использовать переменные экземпляров `public final` (как это сделано в Java для `CASE_INSENSITIVE_ORDER`).

Очереди с приоритетами на основе компараторов

Гибкость, которую обеспечивают компараторы, удобна и при работе с очередями с приоритетами. Для включения компараторов в нашу стандартную реализацию в алгоритме 2.6 понадобится выполнить перечисленные ниже шаги.

- Импортировать библиотеку `java.util.Comparator`.
- Добавить в класс `MaxPQ` переменную экземпляров `comparator` и конструктор, который принимает в качестве аргумента компаратор и присваивает это значение переменной `comparator`.
- Добавить в функцию `less()` проверку, равно ли `null` значение компаратора (с использованием этого значения, если оно не пусто).

После этих добавлений можно, к примеру, создать несколько различных очередей с приоритетами на основе различных ключей из класса `Transaction` — времени, места или номера счета. А если убрать из класса `MaxPQ` фразу `Key extends Comparable<Key>`, то можно работать даже с ключами без естественного порядка.

Устойчивость

Метод сортировки называется *устойчивым*, если он сохраняет относительный порядок одинаковых ключей в массиве (рис. 2.5.1). Часто это свойство бывает важно. Например, рассмотрим приложение электронной коммерции, где нужно обрабатывать большое количество событий с указанием места и времени их возникновения. Допустим, что эти события сохраняются в массиве по мере их поступления — т.е. они уже упорядочены по времени. И допустим, что в приложении нужно распределить события по городам. Простое решение — отсортировать массив по местоположениям. Если сортировка неустойчива, то после ее выполнения транзакции для каждого города *могут* оказаться не упорядоченными по отметкам времени. Программисты, не знакомые с этим, часто удивляются, впервые столкнувшись с такой ситуацией, когда алгоритм перемешивает данные. Некоторые из рассмотренных нами алгоритмов устойчивы (сортировка вставками и слиянием), но большинство таким свойством не обладают (сортировка выбором, Шелла, быстрая и пирамидальная сортировка). Есть способы заставить любую сортировку вести себя устойчиво (см. упражнение 2.5.18), но обычно лучше все-таки использовать устойчивый алгоритм, если устойчивость важна. Однако устойчивость не достигается просто так: все распространенные методы сортировки устойчивы за счет дополнительного расхода времени или памяти (исследователи разработали алгоритмы без этих лишних затрат, однако прикладные программисты сочли их слишком сложными, чтобы быть полезными).

Какой же алгоритм сортировки лучше использовать?

В этой главе мы рассмотрели несколько алгоритмов сортировки, и появление такого вопроса вполне естественно. Решение, какой алгоритм лучше, существенно зависит от деталей приложения и реализации, но мы рассмотрели некоторые методы общего назначения, которые почти абсолютно эффективны в широком спектре приложений.

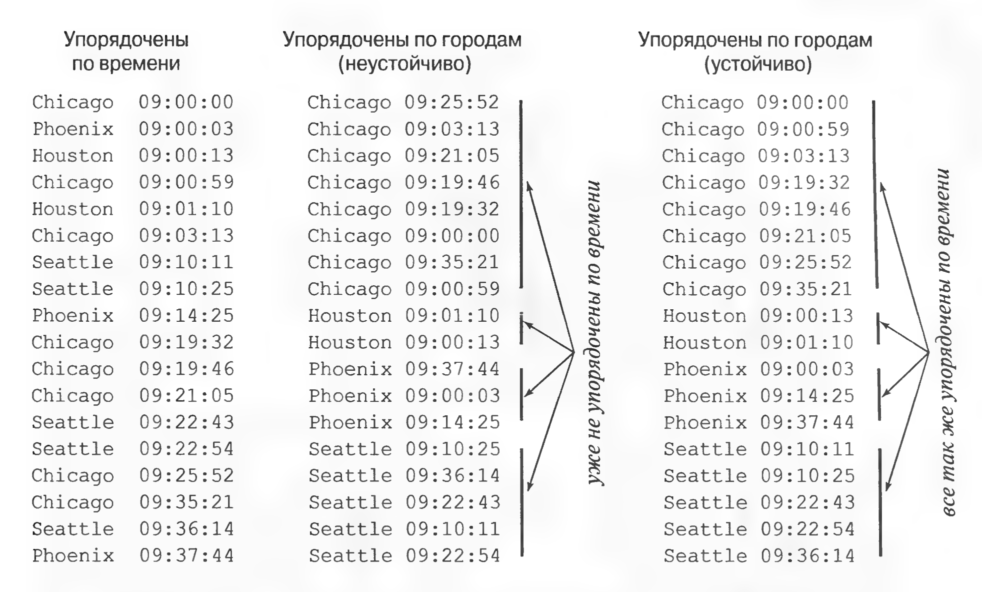


Рис. 2.5.1. Устойчивость при сортировке по второму ключу

Сводкой и общим руководством по важным характеристикам алгоритмов сортировки, которые были рассмотрены в этой главе, может служить табл. 2.5.1. Во всех случаях, кроме сортировки Шелла (где скорость роста оценочная), сортировки вставками (где скорость роста зависит от упорядоченности входных ключей) и обеих версий быстрой сортировки (где скорость роста является вероятностной и может зависеть от исходного распределения ключей), умножение этих скоростей роста на подходящую константу позволяет удобно прогнозировать время выполнения. Эти константы отчасти зависят от алгоритма (например, пирамидальная сортировка выполняет вдвое больше сравнений, чем сортировка слиянием, а они обе выполняют гораздо больше сравнений, чем быстрая сортировка), однако в основном они зависят от реализации, компилятора Java и конкретного компьютера, который определяет скорость выполнения машинных инструкций. Но более важно то, что это константы, и их можно экспериментально определить для небольших N, а затем прогнозировать время выполнения с помощью экстраполяции для больших N, используя наш стандартный способ удвоения.

Утверждение Ф. Быстрая сортировка является наиболее быстрой сортировкой общего назначения.

Обоснование. Эта гипотеза поддерживается бесчисленными реализациями быстрой сортировки на бесчисленных компьютерных системах на протяжении нескольких десятков лет с момента ее изобретения. Основной причиной такой скорострельности быстрой сортировки является то, что ее внутренний цикл содержит лишь несколько инструкций (и хорошо согласуется с кешированием памяти, т.к. чаще всего обращения к данным выполняются последовательно). Поэтому время ее выполнения равно $\sim cN\log N$, где значение c меньше соответствующих констант для других линейно-логарифмических сортировок. А трехчастное разбиение делает быструю сортировку даже линейной для некоторых распределений ключей, которые нередко встречаются на практике — в то время как другие методы сортировки остаются линейно-логарифмическими.

Таблица 2.5.1. Характеристики производительности алгоритмов сортировки

Алгоритм	Устойчивость	На месте	Порядок сортировки роста для N элементов		Примечание
			Время выполнения	Дополнительная память	
Сортировка выбором	нет	да	N^2	1	
Сортировка вставками	да	да	от N до N^2	1	Зависит от порядка элементов
Сортировка Шелла	нет	да	$N\log N$? $N^{6/5}$?	1	
Быстрая сортировка	нет	да	$N\log N$	$\lg N$	Вероятностная гарантия
Трехчастная быстрая сортировка	нет	да	от N до $N\log N$	$\lg N$	Вероятностная гарантия, зависит от распределения ключей
Сортировка слиянием	да	нет	$N\log N$	N	
Пирамидальная сортировка	нет	да	$N\log N$	1	

Поэтому в большинстве практических ситуаций рекомендуется применять быструю сортировку. Однако существует так много применений сортировки и настолько много компьютеров и систем, что простое утверждение вроде этого обосновать трудно. К примеру, мы уже видели одно примечательное исключение: если важна устойчивость и доступна дополнительная память, то может оказаться удобнее использовать сортировку слиянием. Другие исключения будут рассмотрены в главе 5. С помощью средств, подобных `SortCompare`, и при возможности потратить большое количество времени и усилий вы можете выполнить более подробное исследование относительной производительности этих алгоритмов и усовершенствований к ним — как указано в нескольких упражнениях в конце данного раздела. Пожалуй, лучше всего интерпретировать утверждение Ф так: в любом приложении сортировки, где важно время выполнения, в первую очередь следует серьезно обдумать использование быстрой сортировки.

Сортировка примитивных типов

В некоторых приложениях, где важна производительность, может понадобиться сортировка чисел, и может оказаться разумным не использовать ссылки и сортировать непосредственно примитивные типы. Рассмотрим, к примеру, разницу между сортировкой значений `double` и сортировкой значений `Double`. В первом случае обмениваются и упорядочиваются сами числа, а во втором — обмениваются ссылки на объекты `Double`, которые содержат числа. Если нужно просто отсортировать большой массив чисел, мы экономим память для такого же количества ссылок и время на обращения к числам по ссылкам — не говоря уже об отказе от вызовов методов `compareTo()` и `less()`. Для таких случаев можно разработать эффективные версии наших программ сортировки: нужно заменить `Comparable` на имя примитивного типа и переопределить или просто заменить вызовы `less()` выражениями вроде `a[i] < a[j]` (см. упражнение 2.1.26).

Системная сортировка Java

В качестве примера применения информации, приведенной в табл. 2.5.1, рассмотрим основной системный метод сортировки в Java — `java.util.Arrays.sort()`. Учитывая перегрузку типов аргументов, это имя на самом деле означает целый набор методов:

- отдельный метод для каждого примитивного типа;
- метод для типов данных, которые реализуют интерфейс `Comparable`;
- метод, использующий компаратор.

Создатели системы Java выбрали быструю сортировку (с 3-частным разбиением) для сортировки примитивных типов и сортировку слиянием для ссылочных методов. Основным практическим следствием такого выбора является достижение компромисса между затратами времени и памяти (для примитивных типов) и устойчивостью (для ссылочных типов).

Рассмотренные нами алгоритмы и идеи входят в качестве существенной части во многие современные системы, включая и Java. При разработке прикладных Java-программ вы, скорее всего, обнаружите, что одна из реализаций `Arrays.sort()` (возможно, с добавлением вашей реализации метода `compareTo()` и/или `compare()`) вполне годится для ваших нужд, ведь это будет 3-частная быстрая сортировка или сортировка слиянием — проверенные временем классические алгоритмы.

В этой книге в клиентах сортировки мы обычно будем пользоваться нашими собственными методами `Quick.sort()` (как правило) или `Merge.sort()` (если важна устойчивость и хватает памяти). Но вы можете применять `Arrays.sort()` — если нет серьезной причины использовать какой-то другой метод.

Сведения

Принцип, согласно которому алгоритмы сортировки можно применять для решения других задач, является примером базового приема в проектировании алгоритмов, который называется *сведе́нием*. Подробно мы изучим сведение в главе 6 — в силу его важности в теории алгоритмов. А пока мы просто рассмотрим несколько практических примеров. Сведение представляет собой ситуацию, когда алгоритм, разработанный для одной задачи, применяется для решения другой задачи. Прикладные программисты часто пользуются концепцией сведения (хотя это не всегда явно обозначается): каждый раз, когда вы используете метод решения задачи B, чтобы решить задачу A, вы выполняете сведение A к B. И одной из целей при реализации алгоритмов как раз и является облегчение сведения, чтобы алгоритм был применим к как можно более широкому спектру приложений. Вначале мы рассмотрим несколько элементарных примеров сортировки. Многие из них выглядят как алгоритмические головоломки, при решении которых на ум сразу приходит примитивный алгоритм с квадратичным временем выполнения. Часто бывает так, что предварительное упорядочение данных облегчает последующее решение задачи до линейного дополнительного времени — т.е. общая трудоемкость решения снижается с квадратичной до линейной.

Дубликаты

Имеются ли в массиве объектов `Comparable` повторяющиеся ключи? Сколько в нем различных ключей? Какие значения встречаются чаще всего? Для небольших массивов на подобные вопросы нетрудно найти ответы с помощью квадратичного алгоритма, который сравнивает каждый элемент массива с каждым другим элементом массива. Но для больших массивов применение квадратичных алгоритмов неприемлемо. Сортировка позволяет ответить на такие вопросы за линейно-логарифмическое время: нужно сначала упорядочить массив, а потом выполнить проход по нему, отмечая повторяющиеся ключи, которые теперь расположены друг за другом. Например, фрагмент кода, приведенный в листинге 2.5.4, подсчитывает количество различных ключей в массиве. Небольшие изменения в этом коде позволят ответить на все вышеперечисленные вопросы и выполнить задачи вроде вывода всех различных значений, всех повторных значений и т.д. — даже для очень больших массивов.

Листинг 2.5.4. Подсчет различных ключей в массиве `a[]`

```
Quick.sort(a);
int count = 1;  // Предполагается, что a.length > 0.
for (int i = 1; i < a.length; i++)
   if (a[i].compareTo(a[i-1]) != 0)
      count++;
```

Ранжирование

Перестановка (или *расстановка*) — это массив N целых чисел, где каждое из чисел от 0 до $N-1$ встречается ровно один раз. *Расстоянием* (тау) *Кенделла* между двумя расстановками называется количество пар, которые находятся в этих расстановках в разном порядке.

Например, расстояние Кенделла между расстановками `0 3 1 6 2 5 4` и `1 0 3 6 4 2 5` равно четырем, поскольку пары `0-1`, `3-1`, `2-4`, `5-4` находятся в них в разном относительном порядке, а все остальные пары — в том же относительном порядке. Эта статистика

широко применяется в социологии (в теории выборов и голосования), в молекулярной биологии (для сравнения генов с помощью профилей экспрессии), при ранжировании результатов поиска в Интернете, а также во многих других приложениях. Расстояние Кенделла между некоторой расстановкой и тождественной расстановкой (где каждый элемент равен его индексу) равно количеству инверсий в расстановке, и для его подсчета нетрудно сочинить квадратичный алгоритм на основе сортировки вставками (вспомните утверждение В из раздела 2.1). Эффективное вычисление расстояния Кенделла представляет собой интересное упражнение для программиста (или студента), знакомого с описанными у нас классическими алгоритмами сортировки (см. упражнение 2.5.19).

Сведения к очереди с приоритетами

В разделе 2.4 были рассмотрены два примера задач, которые сводятся к последовательности операций над очередями с приоритетами. Программа `TopM` (см. листинг 2.4.1) находит во входном потоке M элементов с наибольшими ключами. Программа `Multiway` (см. листинг 2.4.6) сливает M упорядоченных входных потоков вместе и получает упорядоченный выходной поток. Обе эти задачи легко разрешимы с помощью очереди с приоритетами размером M.

Медиана и порядковые статистики

Важное приложение, связанное с сортировкой, но для которого не нужна полная упорядоченность — операция поиска *медианы* в наборе ключей (такое значение, что половина ключей не больше его, а половина — не меньше, рис. 2.5.2). Эту операцию часто выполняют в статистике и в различных других приложениях обработки данных. Поиск медианы представляет собой частный случай *выбора* — т.е. поиска k-го наименьшего в наборе элементов. Выбор обычно выполняется при обработке экспериментальных и других данных. Определение медианы и других *порядковых статистик* часто применяется для разбиения массива на меньшие группы. Зачастую для дальнейшей обработки нужно сохранить лишь небольшую часть большого массива, и программа, которая способна выбрать, скажем, 10% наибольших элементов, может оказаться удобнее полной сортировки. Наше приложение `TopM` из раздела 2.4 решает эту задачу для входного потока произвольного объема, используя очередь с приоритетами. Если элементы уже находятся в массиве, то эффективной альтернативой `TopM` может быть обычная сортировка: после вызова `Quick.sort(a)` k наименьших значений массива будут находиться в первых k позициях массива. Но сортировка требует линейно-логарифмического времени выполнения, а можно ли обойтись меньшим временем? Поиск k наименьших значений в массиве нетрудно выполнить, если k очень мало или очень велико, но гораздо труднее, если k представляет собой постоянную часть размера массива — например, при поиске медианы ($k = N/2$). Но, как ни удивительно, эту задачу можно решить за

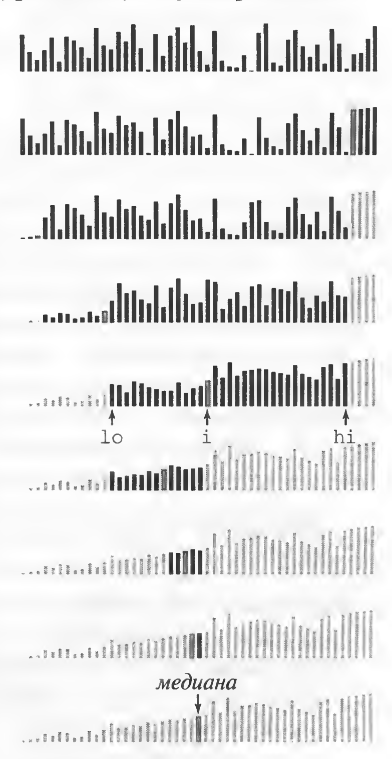


***Рис. 2.5.2.** Поиск медианы с помощью разбиения*

линейное время, как в методе `select()`, приведенном в листинге 2.5.5 (в данной реализации требуется выполнение приведения в клиенте; более аккуратный код без этого ограничения см. на сайте книги). Метод `select()` использует переменные `lo` и `hi`, которые очерчивают подмассив, содержащий индекс `k` нужного элемента, а также использует разбиение из быстрой сортировки для сокращения этого подмассива. Вспомните, что метод `partition()` переупорядочивает элементы от `a[lo]` до `a[hi]` и возвращает целое значение `j`, такое, что элементы с `a[lo]` по `a[j-1]` меньше или равны `a[j]`, а элементы с `a[j+1]` до `a[hi]` больше или равны `a[j]`. Если после этого `k` равно `j`, то все готово. Иначе если `k < j`, необходимо продолжить обработку левого подмассива (изменив значение `hi` на `j-1`), а если `k > j`, необходимо продолжить обработку правого подмассива (изменив значение `lo` на `j+1`). В цикле метода сохраняется инвариант, что слева от `lo` нет элементов, бо́льших, а справа от `hi` нет элементов, меньших любого элемента из подмассива `a[lo..hi]`. После разбиения мы снова сужаем интервал, пока он не будет содержать только `k`. И тогда `a[k]` содержит $(k+1)$-й наименьший элемент, элементы с `a[0]` по `a[k-1]` меньше или равны `a[k]`, а элементы с `a[k+1]` до конца массива больше `a[k]`.

Листинг 2.5.5. Выбор `k` наименьших ключей из массива `a[]`

```
public static Comparable
   select(Comparable[] a, int k)
{
   StdRandom.shuffle(a);
   int lo = 0, hi = a.length - 1;
   while (hi > lo)
   {
      int j = partition(a, lo, hi);
      if      (j == k)  return a[k];
      else if (j > k)  hi = j - 1;
      else if (j < k)  lo = j + 1;
   }
   return a[k];
}
```

Чтобы понять, почему этот алгоритм выполняется за линейное время, предположим, что каждое разбиение делит массив точно пополам. Тогда количество сравнений равно $N + N/2 + N/4 + N/8 + \ldots$ — эта сумма заканчивается там, где найден k-й наименьший элемент, и она меньше $2N$. Как и в случае быстрой сортировки, для нахождения точной границы (которая несколько больше) нужно немного повозиться с математикой. И, как в случае с быстрой сортировкой, анализ зависит от разбиений по случайным элементам, поэтому гарантия получается вероятностной.

Утверждение Х. Выбор на основе разбиения в среднем выполняется за линейное время.

Доказательство. Анализ похож на доказательство утверждения Л для быстрой сортировки (но существенно сложнее) и приводит к результату, что среднее количество сравнений равно $\sim 2N + 2k\ln(N/k) + 2(N-k)\ln(N/(N-k))$, а оно линейно для любого допустимого k. Например, из этой формулы следует, что поиск медианы ($k = N/2$) потребует в среднем $\sim(2 + 2\ln 2)N$ сравнений. Учтите, что худший случай квадратичен, но, как и в случае с быстрой сортировкой, от него защищает рандомизация.

Разработка алгоритма выбора, который гарантированно использует линейное количество сравнений *в худшем случае*, представляет собой классический результат в оценке вычислительной сложности, но он пока не привел к полезному на практике алгоритму.

Краткий обзор применений сортировки

Непосредственные применения сортировки настолько всем хорошо знакомы, вездесущи и многочисленны, что перечислить их просто невозможно. Мы упорядочиваем музыкальные произведения по их названиям и именам исполнителей, почтовые сообщения и телефонные звонки по времени их появления, а фотографии — по датам. В университетах студенты упорядочены по именам или учетным номерам. Банки сортируют миллионы, а то и миллиарды транзакций по датам или суммам. Ученые упорядочивают не только экспериментальные данные (по времени замера или другому параметру), но и результаты подробного моделирования природных данных — от движения элементарных частиц и небесных тел до структуры материалов и социальных отношений. Вообще-то сложно найти область вычислений, в которой *не* применяется сортировка! Поэтому мы в этом разделе рассмотрим примеры применений, более сложные, чем только что рассмотренные сведения, в том числе и те, которые мы изучим подробнее ниже в данной книге.

Коммерческие вычисления

Мир буквально захлебывается информацией. Правительственные организации, финансовые учреждения и коммерческие предприятия выполняют сортировку информации. Если необходимо упорядочить счета по именам или номерам, транзакции по датам или суммам, почту по почтовым кодам или адресам, файлы по именам или датам — и что угодно еще — при обработке таких данных обязательно используется алгоритм сортировки. Обычно подобная информация организуется в виде огромных баз данных, упорядоченных по нескольким ключам для эффективности поиска. При этом часто применяется следующая стратегия: новая информация накапливается, добавляется в базу данных, сортируется по нужным ключам, и затем упорядоченная информация сливается по всем ключам с существующей базой. Изученные нами методы эффективно применялись с самого начала компьютерной обработки данных и позволяют создать огромную инфраструктуру упорядоченных данных, и эти методы лежат в основе всей коммерческой деятельности. В настоящее время повсеместно обрабатываются массивы, содержащие миллионы и даже миллиарды элементов, а без линейно-логарифмических алгоритмов такие массивы невозможно было бы сортировать — т.е. подобная обработка была бы трудна или даже невозможна.

Поиск информации

Хранение данных в упорядоченном виде позволяет эффективно выполнять поиск в них с помощью классического алгоритма *бинарного поиска* (см. главу 1). Вы увидите, что эти же схемы позволяют быстро обслуживать и многие другие виды запросов. Сколько элементов меньше заданного элемента? Какие элементы попадают в указанный диапазон? Такие вопросы мы будем рассматривать в главе 3. Мы также подробно рассмотрим различные расширения сортировки и бинарного поиска, позволяющие перемешивать подобные запросы с операциями, которые вставляют объекты в набор и удаляют из него, все-таки гарантируя логарифмическую производительность для всех операций.

Исследование операций

Исследование операций занимается разработкой и применением математических моделей для решения задач и принятия решений. В этой книге мы увидим несколько примеров взаимосвязи исследования операций с изучением алгоритмов, начиная с применения сортировки в классической задаче — *планировании*. Пусть имеется N заданий, которые нужно выполнить, и задание j требует для своего выполнения t_j секунд времени. Нужно выполнить все задания, минимизировав среднее время выполнения заданий. Известно, что этой цели можно достичь с помощью правила *наименьшее время выполнения — сначала*, т.е. нужно запланировать выполнение заданий в порядке возрастания требуемого для них времени. Значит, можно отсортировать задания по времени выполнения или поместить их в очередь с приоритетами, ориентированную на минимум. При наличии различных других условий и ограничений мы будем получать различные другие задачи планирования, которые часто возникают в производственных приложениях и хорошо изучены.

В качестве еще одного примера можно рассмотреть *задачу балансировки нагрузки*, где имеется M идентичных процессоров и N заданий, которые нужно выполнить. Требуется так спланировать выполнение заданий на процессорах, чтобы последнее задание завершилось как можно раньше. Эта задача является NP-трудной (см. главу 6), поэтому не стоит ожидать, что найдется практический способ для вычисления оптимального графика. Один из методов, который обычно позволяет получить хороший график — правило *наибольшее время выполнения — сначала*, когда задания выбираются в порядке убывания требуемого для них времени и назначаются процессору, который первый станет доступным. Чтобы реализовать этот алгоритм, необходимо сначала упорядочить задания по убыванию времени выполнения. Затем используется очередь с приоритетами из M процессоров, где приоритет вычисляется как сумма времен обработки заданий на данном процессоре. На каждом шаге мы удаляем из очереди процессор с минимальным приоритетом, добавляем к нему новое задание и снова вставляем его в очередь.

Моделирование, управляемое событиями

Во многих научных приложениях применяется моделирование какого-то аспекта реального мира, позволяющее лучше разобраться в нем. До эпохи компьютерных вычислений ученым приходилось довольствоваться математическими моделями, но теперь такие модели дополняются вычислительными моделями. Эффективное выполнение подобного моделирования может само по себе быть сложной задачей, и использование подходящих алгоритмов, несомненно, может означать разницу между возможностью выполнить моделирование за разумное время и вынужденным выбором между неточными результатами или длительным ожиданием точных. В главе 6 будет рассмотрен подробный пример, иллюстрирующий это положение.

Численные расчеты

В научных вычислениях часто важна их *точность*: насколько полученный результат близок к истинному ответу? Точность крайне важна при выполнении миллионов операций с приблизительными значениями, такими как представления вещественных чисел в виде чисел с плавающей точкой, которые используются в компьютерах. В некоторых численных алгоритмах для управления точностью вычислений используются очереди с приоритетами и сортировка. Например, для выполнения численного интегрирования (квадратуры), где нужно вычислить площадь под некоторой кривой, можно использо-

вать очередь с приоритетами, содержащую оценки точности для множества отрезков, на которые разбит весь интервал интегрирования. Процесс извлекает наименее точный отрезок, делит его пополам (получая лучшую точность для этих половинок) и помещает обе половинки назад в очередь — и так, пока не будет достигнута требуемая общая точность.

Комбинаторный поиск

Классическая парадигма в искусственном интеллекте и при работе с трудными задачами — определение множества *конфигураций* с четко определенными *переходами* от одной конфигурации к другой и приоритетами, назначенными этим переходам. Кроме того, задаются *начальная* конфигурация и *конечная* конфигурация (соответствующая решению задачи). Известен *алгоритм A** — процесс решения задач, где начальная конфигурация помещается в очередь с приоритетами, а затем до достижения конечной конфигурации выполняется следующий шаг: из очереди извлекается конфигурация с наибольшим приоритетом, а в очередь заносятся все конфигурации, которые могут быть достигнуты из только что удаленной за один переход (кроме нее самой). Как и моделирование, управляемое событиями, этот процесс прямо создан для очередей с приоритетами, и решение задачи сводится к определению эффективной функции приоритетов. Пример см. в упражнении 2.5.32.

Кроме таких очевидных применений (а мы затронули лишь очень небольшую их часть), сортировка и очереди с приоритетами выступают как важные абстракции в создании алгоритмов, поэтому они часто будут попадаться вам на глаза в данной книге. Ниже мы приведем некоторые примеры их применений в оставшейся части книги. Все эти приложения зависят от эффективной реализации алгоритмов сортировки и типа данных очереди с приоритетами, которые были рассмотрены в этой главе.

Алгоритм Прима и алгоритм Дейкстры

Это классические алгоритмы из главы 4, где речь идет об обработке *графов* — фундаментальной модели для *элементов* и *ребер*, которые соединяют пары элементов. В основе этих и некоторых других алгоритмов лежит *поиск на графе*, где выполняются переходы по ребрам от элемента к элементу. Очереди с приоритетами играют важную роль в операции поиска на графах, что позволяет создавать эффективные алгоритмы.

Алгоритм Крускала

Это еще один классический алгоритм для графов, ребрам которых приписаны веса: в нем обработка ребер зависит от этих весов. Время выполнения данного алгоритма определяется стоимостью сортировки.

Сжатие Хаффмана

Это классический алгоритм *сжатия данных*, в котором выполняется обработка множества элементов с целочисленными весами, где два элемента с наименьшими весами объединяются с получением элемента с весом, равным сумме весов его компонентов. Реализация такой операции очевидна, если привлечь очередь с приоритетами. На сортировке основываются и несколько других схем сжатия данных.

Алгоритмы обработки строк

Они крайне важны в современных программах криптологии и геномики и часто основываются на алгоритмах сортировки (обычно специальных вариантов, предназначен-

ных для сортировки строк — см. главу 5). Например, в главе 6 будут описаны алгоритмы для поиска *самой длинной повторяющейся подстроки* в заданной строке, которые основаны на предварительной сортировке суффиксов строки.

Вопросы и ответы

Вопрос. **Есть ли тип данных очереди с приоритетами в библиотеке Java?**

Ответ. Да — см. `java.util.PriorityQueue`.

Упражнения

2.5.1. Ниже приведена реализация метода `compareTo()` для типа `String`. Как помогает повысить эффективность третья строка?

```
public int compareTo(String that)
{
   if (this == that) return 0;  // вот эта строка
   int n = Math.min(this.length(), that.length());
   for (int i = 0; i < n; i++)
   {
      if      (this.charAt(i) < that.charAt(i)) return -1;
      else if (this.charAt(i) > that.charAt(i)) return +1;
   }
   return this.length() - that.length();
}
```

2.5.2. Напишите программу, которая читает из стандартного ввода список слов и выводит все составные слова, состоящие из двух слов. Например, если в списке имеются слова `пар`, `ковка` и `парковка`, то `парковка` — составное слово.

2.5.3. Проанализируйте следующую реализацию класса, который предназначен для представления бухгалтерских балансов. Почему метод `compareTo()` неправильно реализует интерфейс `Comparable`?

```
public class Balance implements Comparable<Balance>
{
   ...
   private double amount;
   public int compareTo(Balance that)
   {
      if (this.amount < that.amount - 0.005) return -1;
      if (this.amount > that.amount + 0.005) return +1;
      return 0;
   }
   ...
}
```

Опишите способ устранения этой проблемы.

2.5.4. Реализуйте метод `String[] dedup(String[] a)`, который возвращает объекты из `a[]` в упорядоченном виде, без повторяющихся элементов.

2.5.5. Поясните, почему сортировка выбором неустойчива.

2.5.6. Реализуйте рекурсивную версию метода `select()`.

2.5.7. Сколько примерно сравнений понадобится в среднем, чтобы найти наименьший из N элементов с помощью метода `select()`?

2.5.8. Напишите программу `Frequency`, которая читает строки из стандартного ввода и выводит количество повторений каждой строки, по убыванию частот.

2.5.9. Разработайте тип данных, позволяющий написать клиент, который может упорядочить файл вроде приведенного на рис. 2.5.3.

Входные данные (индексы Доу-Джонса в различные дни)

```
 1-Oct-28     3500000
 2-Oct-28     3850000
 3-Oct-28     4060000
 4-Oct-28     4330000
 5-Oct-28     4360000
      ...
30-Dec-99   554680000
31-Dec-99   374049984
 3-Jan-00   931800000
 4-Jan-00  1009000000
 5-Jan-00  1085500032
      ...
```

Выходные данные

```
19-Aug-40  130000
26-Aug-40  160000
24-Jul-40  200000
10-Aug-42  210000
23-Jun-42  210000
...
23-Jul-02  2441019904
17-Jul-02  2566500096
15-Jul-02  2574799872
19-Jul-02  2654099968
24-Jul-02  2775559936
```

Рис. 2.5.3. Пример данных для упражнения 2.5.9

2.5.10. Создайте тип данных `Version`, который представляет номер версии ПО вроде `115.1.1`, `115.10.1`, `115.10.2`. Реализуйте для него интерфейс `Comparable`, так, чтобы версия `115.1.1` была меньше, чем `115.10.1`.

2.5.11. Один из способов описать результат алгоритма сортировки — указание перестановки `p[]` чисел от `0` до `a.length-1`, такой, что элемент `p[i]` указывает, куда попадает ключ, первоначально бывший в `a[i]`. Приведите перестановки, описывающие результаты сортировки вставками, сортировки выбором, сортировки Шелла, сортировки слиянием, быстрой сортировки и пирамидальной сортировки для массива из семи одинаковых ключей.

Творческие задачи

2.5.12. *Планирование.* Напишите программу `SPT.java`, которая читает из стандартного ввода имена заданий и времена из выполнения и выводит расписание, минимизирующее среднее время выполнения с помощью правила “наименьшее время выполнения — сначала”, описанное в разделе “Исследование операций”.

2.5.13. *Балансировка нагрузки.* Напишите программу LPT.java, которая принимает из командной строки целое число M, читает из стандартного ввода имена заданий и времена их выполнения и выводит расписание назначения заданий M процессорам, которое приблизительно минимизирует время завершения последнего задания с помощью правила "наибольшее время выполнения — сначала", описанное в разделе "Исследование операций".

2.5.14. *Сортировка доменных имен.* Напишите тип данных Domain, представляющий доменные имена и содержащий метод compareTo(), который реализует естественный для доменных имен порядок *обратных* имен. Например, для домена cs.princeton.edu обратным именем будет edu.princeton.cs. Это удобно для анализа журналов работы в Интернете. *Совет*: для разбиения строки s на фрагменты, разделяемые точками, используйте вызов s.split("\\."). Напишите клиент, который читает имена доменов из стандартного ввода и выводит упорядоченный список обратных доменных имен.

2.5.15. *Рассылка спама.* Для рассылки нелегального спама подготовлен список почтовых адресов из различных доменов (часть почтового адреса после символа @). Для фальсификации обратных адресов нужно сделать так, чтобы сообщения якобы посылались от другого пользователя из этого же домена. Например, письмо может быть отправлено с адреса wayne@princeton.edu на адрес rs@princeton.edu. Как нужно подготовить список почтовых адресов, чтобы эффективно выполнить эту задачу?

2.5.16. *Выборы без предпочтений.* Чтобы не ущемлять кандидатов, имена которых находятся в конце алфавита, на выборах губернатора Калифорнии в 2003 г. их упорядочили с помощью следующего набора символов:

```
R W Q O J M V A H B S G Z X N T C I E K U P D Y F L
```

Создайте тип данных, где этот порядок является естественным, и напишите клиент California с единственным статическим методом main(), который упорядочивает строки в таком порядке. Считайте, что все строки содержат только прописные буквы.

2.5.17. *Проверка устойчивости.* Расширьте метод check() из упражнения 2.1.16, чтобы он вызывал метод sort() для указанного массива и возвращал true, если метод sort() выполняет *устойчивое* упорядочение, и false в противном случае. Не считайте, что метод sort() может перемещать данные только с помощью метода exch().

2.5.18. *Обеспечение устойчивости.* Напишите метод-оболочку, который делает устойчивой любую сортировку. Для этого он создает новый тип ключа, составленный из старого ключа и его индекса в массиве, вызывает метод sort(), а затем восстанавливает исходные ключи.

2.5.19. *Расстояние Кенделла.* Напишите программу KendallTau.java, которая вычисляет расстояние Кенделла между двумя расстановками за линейно-логарифмическое время.

2.5.20. *Время простоя.* Пусть компьютер с параллельным процессором выполняет N заданий. Напишите программу, которая по заданным моментам запуска и завершения заданий находит максимальный интервал, когда процессор находился

в состоянии простоя, и максимальный интервал, когда процессор находился в состоянии работы.

2.5.21. *Многомерная сортировка*. Напишите тип данных `Vector` для использования в методах сортировки многомерных векторов, состоящих из *d* целых чисел. Векторы упорядочиваются по первому компоненту, векторы с одинаковыми первыми компонентами — по второму компоненту, с одинаковыми вторыми компонентами — по третьему компоненту и т.д.

2.5.22. *Игра на бирже*. Инвесторы покупают и продают акции предприятий на электронной бирже, указывая максимальную цену покупки и минимальную цену продажи, на которые они согласны, и количество акций, которые они готовы купить или продать по указанной цене. Напишите программу, которая использует очереди с приоритетами для сведения продавцов с покупателями, и проверьте ее работу методом моделирования. Используйте две очереди с приоритетами — одну для покупателей, а другую для продавцов — и выполняйте покупки-продажи, когда новый ордер можно удовлетворить с помощью существующего ордера (ордеров).

2.5.23. *Использование выборки для выбора*. Рассмотрите идею использования выборки для усовершенствования выбора. *Совет*: использование медианы помогает не всегда.

2.5.24. *Устойчивая очередь с приоритетами*. Разработайте *устойчивую* реализацию очереди с приоритетами (которая возвращает одинаковые ключи в порядке их вставки).

2.5.25. *Точки на плоскости*. Напишите три статических компаратора для типа данных `Point2D` (см. рис. 1.2.7): один для сравнения точек по координате *x*, другой — для сравнения по координате *y*, и третий — для сравнения по их расстоянию от начала координат. Напишите два нестатических компаратора для типа данных `Point2D`: один для сравнения по расстоянию от заданной точки, а другой — для сравнения по полярному углу относительно заданной точки.

2.5.26. *Простой многоугольник*. Пусть имеются *N* точек на плоскости; начертите простой (без самопересечений) многоугольник с этими точками в качестве вершин. *Совет*: найдите точку *p* с наименьшей координатой *y*, а если таких несколько, то с наименьшей координатой *x*. Соедините точки по возрастанию их полярного угла относительно *p*.

2.5.27. *Сортировка параллельных массивов*. При сортировке параллельных массивов удобно иметь версию процедуры сортировки, которая возвращает перестановку (например, `index[]`) индексов, упорядочивающую массив. Добавьте в программу `Insertion` метод `indirectSort()`, который принимает в качестве аргумента массив `a[]` объектов `Comparable`, но вместо переупорядочения элементов `a[]` возвращает целочисленный массив `index[]` — такой, что элементы с `a[index[0]]` по `a[index[N-1]]` упорядочены по возрастанию.

2.5.28. *Сортировка файлов по именам*. Напишите программу `FileSorter`, которая принимает из командной строки имя каталога и выводит имена всех находящихся там файлов, упорядоченные по алфавиту. *Совет*: используйте тип данных `File`.

2.5.29. *Сортировка файлов по размеру и дате последнего изменения*. Напишите компараторы для типа `File`, которые позволяют сортировать файлы по возрастанию/

убыванию их размера, по возрастанию/убыванию их имен и по возрастанию/убыванию их даты последнего изменения. Используйте эти компараторы в программе `LS`, которая принимает из командной строки имя каталога и выводит имена всех находящихся там файлов, упорядоченные по заданному критерию — например, по отметке времени при наличии параметра `"-t"`. Программа должна поддерживать несколько параметров для более точного упорядочения и использовать устойчивую сортировку.

2.5.30. *Теорема Бернера.* Правда ли, что если отсортировать каждый столбец матрицы, а потом — каждую строку, то столбцы останутся упорядоченными? Обоснуйте свой ответ.

Эксперименты

2.5.31. *Дубликаты.* Напишите клиент, который принимает из командной строки целочисленные аргументы `M`, `N` и `T`, а затем использует код, приведенный в тексте, чтобы выполнить T повторений следующего эксперимента: генерируется N случайных значений `int` от 0 до $M-1$ и подсчитывается количество повторяющихся. Выполните программу для $T = 10$ и $N = 10^3$, 10^4, 10^5 и 10^6 при $M = N/2$, N и $2N$. Теория вероятности утверждает, что количество дубликатов должно быть примерно $(1-e^{-\alpha})$, где $\alpha = N/M$ — напечатайте таблицу, которая поможет проверить, согласуются ли ваши эксперименты с этой формулой.

2.5.32. *Игра в 8 из 9.* Эта игра была популяризована С. Лойдом (S. Loyd) в 1870-х годах. На квадратной сетке 3×3 расположены 8 фишек, пронумерованных от 1 до 8; одна ячейка оставлена пустой. Нужно передвинуть фишки так, чтобы они были расположены по порядку. Можно двигать любую из фишек по горизонтали или вертикали (но не по диагонали) на пустое место. Напишите программу, которая решает задачу с помощью *алгоритма A**. Сначала определите приоритет как количество ходов, необходимых для получения данной позиции, плюс количество фишек не на своих местах. (Кстати, количество ходов, необходимых для данной позиции, не меньше, чем количество фишек не на своих местах.) Рассмотрите другие функции для количества фишек не на своих позициях — например, сумма расстояний Манхэттена от каждой фишки до ее позиции или сумма квадратов этих расстояний.

2.5.33. *Случайные транзакции.* Напишите генератор, который принимает в качестве аргумента число N и генерирует N случайных объектов `Transaction` (см. упражнения 2.1.21 и 2.1.22), отбирая только "разумные" транзакции. Затем сравните производительность сортировки Шелла, сортировки слиянием, быстрой сортировки и пирамидальной сортировки при упорядочении N транзакций, для $N = 10^3$, 10^4, 10^5 и 10^6.

Глава 3
Поиск

Современные вычисления и Интернет предоставляют доступ к ошеломляющему объему информации. Залогом ее успешной обработки является возможность эффективного поиска в этой информации. В данной главе описываются классические алгоритмы поиска, которые доказали свою эффективность в многочисленных приложениях на протяжении десятков лет. Без подобных алгоритмов развитие вычислительной инфраструктуры, которой мы пользуемся в современном мире, было бы невозможно.

Мы используем термин *таблица имен* (symbol table) для описания абстрактного механизма, который позволяет сохранить информацию (*значение*), а затем найти и выбрать ее, указав *ключ*. Природа ключей и значений зависит от приложения. Поскольку может быть очень много ключей и огромный объем хранимой информации, реализация эффективной таблицы имен представляет собой сложную вычислительную задачу.

Таблицы имен иногда называются *словарями*, по аналогии с проверенной временем системой хранения определений слов с помощью перечисления их в алфавитном порядке в специальном справочнике. В толковом словаре ключом является слово, а его значением — статья, связанная с этим словом и содержащая его определение, произношение и этимологию. Кроме того, таблицы имен иногда называются *индексами*, по аналогии с еще одной привычной системой перечисления терминов в алфавитном порядке в конце учебника. В индексе ключом является термин, а значением — список номеров страниц, где этот термин упоминается.

После описания базовых API и двух фундаментальных реализаций мы рассмотрим три классических структуры данных, которые способны поддерживать эффективные реализации таблиц имен — бинарные деревья поиска, красно-черные деревья и хеш-таблицы. В завершение этой темы мы ознакомимся с несколькими расширениями и применениями, многие из которых были бы невозможны без эффективных алгоритмов, о которых вы узнаете в данной главе.

3.1. Таблицы имен

Основное назначение таблицы имен — связывание *значения* с *ключом*. Клиент может *вставлять* пары ключ-значение в таблицу имен, ожидая, что позже он может выполнить в этой таблице *поиск* значения, связанного с указанным ключом, среди всех пар ключ-значение, которые были помещены в таблицу. В этой главе описано несколько способов структурирования этих данных, которые позволяют эффективно выполнять не только операции *вставить* и *найти*, но и несколько других удобных операций. Для реализации таблицы имен необходимо определить базовую структуру данных, а затем разработать алгоритмы для вставки, поиска и других операций, которые создают и изменяют структуру данных.

Поиск так важен в столь многих компьютерных приложениях, что таблицы имен доступны в виде высокоуровневых абстракций во многих средах программирования, включая и Java (реализации таблиц имен, доступные в Java, будут обсуждаться в разделе 3.5). В табл. 3.1.1 приведено несколько примеров ключей и значений, которые могут понадобиться в типовых приложениях. Вскоре мы рассмотрим некоторые типичные клиенты, а раздел 3.5 будет посвящен эффективному использованию таблиц имен в ваших собственных клиентах. Кроме того, таблицы имен будут применяться в этой книге при разработке других алгоритмов.

Таблица 3.1.1. Типичные применения таблиц имен

Применение	Цель поиска	Ключ	Значение
Словарь	Поиск толкования	Слово	Толкование
Индекс в книге	Поиск релевантных страниц	Термин	Список номеров страниц
Общий каталог	Поиск песни для загрузки	Название песни	Идентификатор компьютера
Управление учетными записями	Обработка транзакций	Номер учетной записи	Детали транзакции
Веб-поиск	Поиск релевантных веб-страниц	Ключевое слово	Список адресов страниц
Компилятор	Поиск типа и значения	Имя переменной	Тип и значение

Определение. *Таблица имен* — это структура данных для хранения пар ключ-значение, которая поддерживает две операции: *вставить* (поместить) новую пару в таблицу и *найти* (получить) значение, связанное с заданным ключом.

API

Таблица имен является прототипным *абстрактным типом данных* (см. главу 1): это набор значений и операций над этими значениями, что позволяет разделить разработку клиентов и реализаций. Как обычно, мы четко определяем операции с помощью интерфейса прикладного программирования (API), который предоставляет контракт между клиентом и реализацией (рис. 3.1.1).

public class **ST**<Key, Value>	
ST()	*создание таблицы имен*
void put(Key key, Value val)	*помещение пары ключ-значение в таблицу (удаление ключа* key *из таблицы, если значение равно* null*)*
Value get(Key key)	*значение, связанное с ключом* key *(*null*, если ключ отсутствует)*
void delete(Key key)	*удаление ключа* key *(и его значения) из таблицы*
boolean contains(Key key)	*имеется ли значение, связанное с ключом* key*?*
boolean isEmpty()	*пуста ли таблица?*
int size()	*количество пар ключ-значение в таблице*
Iterable<Key> keys()	*все ключи из таблицы*

Рис. 3.1.1. *API для обобщенной базовой таблицы имен*

Прежде чем начать рассматривать код клиентов, мы познакомимся с некоторыми вариантами реализации этого API, которые позволяют писать понятный, компактный и полезный код.

Обобщенные типы

Как и в случае сортировки, мы будем рассматривать методы без указания типов элементов, а с применением обобщенных типов. Для таблиц имен мы подчеркнем различие ролей ключей и значений для поиска, явно указывая их типы, а не считая ключи неявно определенными в элементах, как это было в очередях с приоритетами в разделе 2.4. После знакомства с некоторыми характеристиками этого базового API (например, обратите внимание, что упорядоченность ключей не упоминается), мы рассмотрим расширение для типичного случая: ключи реализуют интерфейс Comparable, который позволяет использовать ряд дополнительных методов.

Повторяющиеся ключи

Во всех наших реализациях приняты следующие соглашения.

- С каждым ключом связано лишь одно значение (таблица не содержит повторяющихся ключей).
- Если клиент помещает пару ключ-значение в таблицу, в которой уже есть такой ключ (и связанное с ним значение), новое значение заменяет старое.

Эти соглашения определяют *абстракцию ассоциативного массива*, которая позволяет рассматривать таблицу имен просто как массив, где ключи являются индексами, а значения — элементами массива. В обычном массиве ключами являются целочисленные индексы, которые используются для быстрого доступа к значениям массива; в ассоциативном массиве (таблица имен) ключи имеют произвольный тип, но все равно позволяют быстро обращаться к значениям. В некоторых языках программирования (не Java) встроена специальная поддержка, которая позволяет программистам использовать код вроде st[key] вместо st.get(key) и st[key] = val вместо st.put(key, val), где key и val — объекты произвольного типа.

Нулевые ключи

Ключи не должны быть равны `null`. Как и во многих других механизмах Java, использование нулевого ключа приводит к возникновению исключения во время выполнения (см. третий вопрос в разделе "Вопросы и ответы").

Нулевые значения

Мы также считаем, что ни один ключ не может быть связан со значением `null`. Это соглашение непосредственно связано со спецификацией в API, что функция `get()` должна возвращать `null` для отсутствующих в таблице ключей — т.е., по сути, значение `null` связывается с любым ключом, которого нет в таблице. Из этого соглашения есть два (преднамеренных) следствия. Во-первых, можно проверить, содержит ли таблица имен значение, связанное с данным ключом, проверив, возвращает ли `null` функция `get()`. И, во-вторых, для реализации удаления можно воспользоваться вызовом `put()` со вторым аргументом, равным `null` (как описано в следующем абзаце).

Удаление

Удаление в таблицах имен обычно выполняется с помощью одной из двух стратегий: *ленивое* удаление, когда ключ в таблице связывается со значением `null` (возможно, все такие ключи будут удалены в дальнейшем), и *энергичное* удаление, когда ключ удаляется из таблицы сразу же. Как только что было сказано, код `put(key, null)` представляет собой простую (ленивую) реализацию операции `delete(key)`. Энергичная реализация `delete()` должна действовать по-другому. В наших реализациях таблиц имен, где не применяется ленивая операция `delete()`, реализации `put()` на сайте книги начинаются с защитного кода

```
if (val == null) { delete(key); return; }
```

чтобы ни один ключ в таблице не был связан с `null`. В книге этот код для экономии места опущен (а в клиентском коде нет вызовов `put()` с нулевым аргументом).

Сокращенные методы

Чтобы код клиентов был понятнее, мы включили в API методы `contains()` и `isEmpty()` со стандартными однострочными реализациями, приведенными в табл. 3.1.2. Для экономии места мы не будем повторять этот код и будем просто предполагать, что он присутствует во всех реализациях API таблиц имен, и свободно использовать их в клиентском коде.

Таблица 3.1.2. Стандартные реализации

Метод	Стандартная реализация
`void delete(Key key)`	`put(key, null);`
`boolean contains(key)`	`return get(key) != null;`
`boolean isEmpty()`	`return size() == 0;`

Перебор

Чтобы клиенты могли обрабатывать все ключи и значения из таблицы, в первую строку API можно добавить фразу `implements Iterable<Key>`, означающую, что каждая реализация должна содержать метод `iterator()`, который возвращает итератор с

соответствующими реализациями методов `hasNext()` и `next()`, как это было сделано для очередей и стеков в разделе 1.3. Для таблиц имен мы будем придерживаться более простого подхода: мы определим метод `keys()`, который возвращает клиенту объект `Iterable<Key>`, позволяющий перебрать все ключи. Это связано с необходимостью согласования с методами, которые будут определены для упорядоченных таблиц имен, где клиенты могут выполнить перебор заданного подмножества ключей из таблицы.

Равенство ключей

Определение, находится ли указанный ключ в таблице имен, основано на концепции *равенства объектов*, которая была основательно рассмотрена в разделе 1.2 (подраздел "Равенство"). По принятому в Java соглашению все объекты наследуют метод `equals()` и его реализацию как для стандартных типов вроде `Integer`, `Double` и `String`, так и для более сложных типов вроде `File` и `URL`: при работе с этими типами можно пользоваться их встроенными реализациями. Например, если `x` и `y` — значения типа `String`, то выражение `x.equals(y)` равно `true` тогда и только тогда, когда `x` и `y` имеют одинаковую длину и совпадают в каждой символьной позиции. Для ключей, определенных в клиенте, необходимо переопределить метод `equals()`, как описано в разделе 1.2. Нашу реализацию метода `equals()` для типа `Date` (см. листинг 1.2.6) можно использовать в качестве шаблона для разработки аналогичных методов для наших собственных типов. Как было указано для очередей с приоритетами, лучше делать типы `Key` неизменными, т.к. иначе невозможно гарантировать согласованность информации.

Упорядоченные таблицы имен

В типичных приложениях ключи являются объектами `Comparable`, поэтому для сравнения двух ключей `a` и `b` можно использовать код `a.compareTo(b)`. Упорядоченность ключей, которая вытекает из интерфейса `Comparable`, используется в нескольких реализациях таблиц имен для эффективного выполнения операций `put()` и `get()`. Более важно то, что в таких реализациях можно считать, что таблица имен *хранит упорядоченные ключи*, и значительно расширить API, определив множество естественных и полезных операций, учитывающих относительный порядок ключей. Пусть, например, ключи представляют собой время дня. Может понадобиться узнать самый ранний или поздний момент, множество ключей, которые попадают между двумя заданными моментами, и т.д. В большинстве случаев подобные операции нетрудно реализовать с помощью тех же структур данных и методов, на которых основаны реализации `put()` и `get()`. А именно, для приложений с ключами `Comparable` мы реализуем в данной главе API, приведенный на рис. 3.1.2 (см. также рис. 3.1.3).

Признаком, что одна из наших программ реализует этот API, является наличие конструкции `Key extends Comparable<Key>` с описанием переменной обобщенного типа, которая указывает, что код использует свойство ключей `Comparable` и реализует более широкий спектр операций. Все вместе эти операции определяют для клиентских программ *упорядоченную таблицу имен*.

Минимум и максимум

Пожалуй, наиболее естественные запросы к набору упорядоченных ключей — запросы наименьшего и наибольшего ключей. Эти операции уже встречались нам при обсуждении очередей с приоритетами в разделе 2.4.

public class **ST**<Key extends Comparable<Key>, Value>	
ST()	*создание упорядоченной таблицы имен*
void put(Key key, Value val)	*помещение пары ключ-значение в таблицу (удаление ключа* `key` *из таблицы, если значение равно* `null`*)*
Value get(Key key)	*значение, связанное с ключом* `key` *(*`null`*, если он отсутствует)*
void delete(Key key)	*удаление ключа* `key` *(и его значения) из таблицы*
boolean contains(Key key)	*имеется ли значение, связанное с ключом* `key`*?*
boolean isEmpty()	*пуста ли таблица?*
int size()	*количество пар ключ-значение*
Key min()	*наименьший ключ*
Key max()	*наибольший ключ*
Key floor(Key key)	*наибольший ключ, меньший или равный* `key`
Key ceiling(Key key)	*наименьший ключ, больший или равный* `key`
int rank(Key key)	*количество ключей, меньших* `key`
Key select(int k)	*ключ ранга* `k`
void deleteMin()	*удаление наименьшего ключа*
void deleteMax()	*удаление наибольшего ключа*
int size(Key lo, Key hi)	*количество ключей в интервале* `[lo..hi]`
Iterable<Key> keys(Key lo, Key hi)	*упорядоченные ключи из интервала* `[lo..hi]`
Iterable<Key> keys()	*упорядоченные ключи из всей таблицы*

Рис. 3.1.2. *API для обобщенной упорядоченной таблицы имен*



Рис. 3.1.3. *Примеры операций в упорядоченной таблице имен*

В упорядоченных таблицах имен также имеются методы для удаления максимального и минимального ключей (и связанных с ними значений). Они позволяют таблице имен выполнять действия, характерные для класса `IndexMinPQ`, который был рассмотрен в разделе 2.4. Но есть и существенные отличия: в очередях с приоритетами одинаковые ключи разрешены, а в таблицах имен — нет, и упорядоченные таблицы имен поддерживают гораздо больший набор операций.

Верхняя и нижняя опоры

Часто бывает нужно найти для заданного значения *нижнюю опору* (наибольший ключ, который меньше или равен этому значению) и *верхнюю опору* (наименьший ключ, который больше или равен этому значению). Названия соответствующих функций `floor` и `ceiling` взяты из функций работы с вещественными числами: `floor` (пол) — это наибольшее целое число, которое меньше или равно аргументу, а `ceiling` (потолок) — наименьшее целое число, которое больше или равно аргументу.

Ранг и выбор

Базовыми операциями для определения, не нарушает ли новый ключ упорядоченность, являются операции *ранга* (`rank` — определение количества ключей, меньших данного ключа) и *выбора* (`select` — поиск ключа с заданным рангом). Это обратные друг другу действия: для всех `i` от `0` до `size()-1` верно, что `i == rank(select(i))`, и для всех ключей из таблицы верно, что `key == select(rank(key))`.

О необходимости этих операций мы уже говорили при обсуждении применений сортировки в разделе 2.5. В таблицах имен необходимо выполнять эти операции быстро, да еще чередуя со вставками, удалениями и поисками.

Запросы диапазонов

Сколько ключей попадает в указанный диапазон (между двумя заданными ключами)? Какие именно ключи попадают в него? Ответы на этот вопрос должны давать методы `size()` и `keys()` с двумя аргументами, которые часто бывают нужны, особенно в больших базах данных. Необходимость выполнения таких запросов является одной из основных причин широкого применения таблиц имен.

Исключительные случаи

Если метод должен возвратить ключ, а в таблице нет ключа, соответствующего описанию, методы должны генерировать исключение (хотя можно просто возвращать `null`). Например, методы `min()`, `max()`, `deleteMin()`, `deleteMax()`, `floor()` и `ceiling()` должны генерировать исключения, если таблица пуста, а вызов `select(k)` — если `k` меньше `0` или не меньше `size()`.

Комбинированные операции

Как мы уже видели на примере методов `isEmpty()` и `contains()`, в нашем базовом API имеются формально лишние методы, которые позволяют сделать код клиента более простым и понятным. Для экономии места мы предполагаем (если не указано обратное), что все реализации API упорядоченной таблицы имен содержат стандартные реализации, приведенные в табл. 3.1.3.

Таблица 3.1.3. Стандартные реализации дополнительных методов для упорядоченной таблицы имен

Метод	Стандартная реализация
`void deleteMin()`	`delete(min());`
`void deleteMax()`	`delete(max());`
`int size(Key lo, Key hi)`	`if (hi.compareTo(lo) < 0)` `    return 0;` `else if (contains(hi))` `    return rank(hi) - rank(lo) + 1;` `else` `    return rank(hi) - rank(lo);`
`Iterable<Key> keys()`	`return keys(min(), max());`

Еще раз о равенстве ключей

В Java рекомендуется иметь согласованные определения методов compareTo() и equals() для всех типов Comparable. То есть для любой пары значений a и b из любого заданного типа Comparable выражения a.compareTo(b) == 0 и a.equals(b) должны иметь одинаковые значения. Во избежание любых возможных неясностей мы не будем использовать метод equals() в реализациях упорядоченных таблиц имен. Для сравнения ключей будет применяться только метод compareTo(), и для ответа на вопрос, равны ли a и b, мы будем использовать логическое выражение a.compareTo(b) == 0. Обычно такая проверка будет завершать успешный поиск значения a в таблице имен (когда найден элемент b). Как мы видели при изучении алгоритмов сортировки, в Java имеются стандартные реализации compareTo() для многих часто встречающихся типов ключей, и нетрудно разработать реализацию для произвольного собственного типа данных (см. раздел 2.5).

Модель стоимости

И при использовании метода equals() (для таблиц имен с ключами, не реализующими интерфейс Comparable), и при использовании метода compareTo() (для упорядоченных таблиц имен с ключами Comparable) мы будем применять термин *сравнение* для операции сравнения элемента таблицы имен и ключа поиска. В большинстве реализаций таблиц имен эта операция находится во внутреннем цикле. В отдельных случаях, когда это не так, мы будем учитывать и обращения к массиву.

Модель стоимости поиска

При изучении реализаций таблиц имен мы считаем *сравнения* (проверки на равенство или сравнения ключей). В (редких) случаях, когда сравнения не присутствуют во внутреннем цикле, мы подсчитываем *обращения к массиву*.

Реализации таблиц имен обычно различаются по их базовым структурам данных и реализациям методов get() и put(). Мы не всегда будем приводить в тексте реализации остальных методов, т.к. многие из них представляют собой хорошие упражнения на понимание базовых структур данных. Для различения реализаций мы будем добавлять к ST в имени класса префикс, который соответствует реализации символьной таблицы. Для обращения к эталонной реализации в клиентах мы будем писать просто ST, за исключением случаев, когда нужна какая-то конкретная реализация. Постепенно у вас сформируется понимание смысла всех методов из API в контексте различных клиентов

и реализаций таблиц имен, которые будут представлены и рассмотрены в данной главе и в остальной части книги. Кроме того, в разделе "Вопросы и ответы" и в упражнениях будут рассмотрены дополнительные альтернативы и варианты построения.

Примеры клиентов

Подробное описание применений поиска будет приведено в разделе 3.5, но прежде чем приступить к изучению реализаций, неплохо рассмотреть и примеры клиентского кода. Так что сейчас мы рассмотрим два клиента: клиент тестирования, который будет использоваться для трассировки поведения алгоритмов на входных данных небольшого объема, и клиент производительности, который будет стимулировать разработку эффективных реализаций.

Клиент тестирования

Для трассировки поведения наших алгоритмов на небольших входных данных мы будем использовать клиент тестирования, код которого приведен в листинге 3.1.1. Он принимает из стандартного ввода последовательность строк, создает таблицу имен, которая связывает значение `i` с `i`-й входной строкой, а затем выводит полученную таблицу. В приведенных далее трассировках мы будем считать, что входные данные представляют собой односимвольные строки — чаще всего последовательность `S E A R C H E X A M P L E` (рис. 3.1.4). По нашему соглашению, клиент присваивает ключу `S` значение `0`, ключу `R` — значение `3` и т.д. Однако строке `E` будет присвоено значение `12` (но не `1` и не `6`), а строке `A` — значение `8` (но не `2`), поскольку по еще одному соглашению с каждым ключом связывается значение из самого последнего вызова `put()`. Для простейших (неупорядоченных) реализаций порядок ключей в выходных данных этого клиента тестирования не определен (зависит от характеристик реализации), а для упорядоченных таблиц имен клиент тестирования выводит ключи в отсортированном виде. Данный клиент представляет собой пример клиента *индексации* — специального случая фундаментального применения таблиц имен, о котором будет рассказано в разделе 3.5.

Листинг 3.1.1. Базовый клиент тестирования для таблиц имен

```
public static void main(String[] args)
{
   ST<String, Integer> st;
   st = new ST<String, Integer>();
   for (int i = 0; !StdIn.isEmpty(); i++)
   {
      String key = StdIn.readString();
      st.put(key, i);
   }
   for (String s : st.keys())
      StdOut.println(s + " " + st.get(s));
}
```

Клиент производительности

Программа `FrequencyCounter` (см. листинг 3.1.2) — это клиент таблицы имен, который подсчитывает количество появлений каждой строки (с длиной не меньше заданного предела) в последовательности строк из стандартного ввода, а затем перебирает ключи, чтобы найти тот, который встречался чаще всего.

Ключи	S	E	A	R	C	H	E	X	A	M	P	L	E
Значения	0	1	2	3	4	5	6	7	8	9	10	11	12

Выходные данные для обычной таблицы имен (один из вариантов)		Выходные данные для упорядоченной таблицы имен	
L	11	A	8
P	10	C	4
M	9	E	12
X	7	H	5
H	5	L	11
C	4	M	9
R	3	P	10
A	8	R	3
E	12	S	0
S	0	X	7

Рис. 3.1.4. *Ключи, значения и результат работы для клиента тестирования*

Этот клиент является примером клиента *словаря* — приложения, о котором мы поговорим подробнее в разделе 3.5. Такой клиент отвечает на простой вопрос: какое слово (не короче заданной длины) чаще всего встречается в тексте? В данной главе мы будем измерять производительность этого клиента на трех стандартных входных файлах: первые пять строк романа Ч. Диккенса *Tale of Two Cities* ("Повесть о двух городах", `tinyTale.txt`), полный текст этой книги (`tale.txt`) и популярный сборник из 1 миллиона предложений, случайно взятых из Интернета — *Leipzig Corpora Collection* (`leipzig1M.txt`). Содержимое `tinyTale.txt` приведено в листинге 3.1.3.

Листинг 3.1.2. Клиент таблицы имен

```
public class  FrequencyCounter
{
  public static void main(String[] args)
  {
    int minlen = Integer.parseInt(args[0]);   // Длина отбрасываемых ключей
    ST<String, Integer> st = new ST<String, Integer>();
    while (!StdIn.isEmpty())
    {  // Создание таблицы имен и подсчет частот.
       String word = StdIn.readString();
       if (word.length() < minlen) continue;  // Короткие ключи игнорируются.
       if (!st.contains(word)) st.put(word, 1);
       else                    st.put(word, st.get(word) + 1);
    }
    // Поиск ключа с максимальным счетчиком повторений.
    String max = "";
    st.put(max, 0);
    for (String word : st.keys())
       if (st.get(word) > st.get(max))
          max = word;
    StdOut.println(max + " " + st.get(max));
```

Клиент ST подсчитывает частоту появления строк в стандартном вводе, а затем выводит строку с максимальной частотой. В аргументе командной строки задается нижняя граница длины рассматриваемых ключей.

```
% java FrequencyCounter 1 < tinyTale.txt
it 10

% java FrequencyCounter 8 < tale.txt
business 122

% java FrequencyCounter 10 < leipzig1M.txt
government 24763
```

Листинг 3.1.3. Небольшой тестовый файл

```
% more tinyTale.txt
it was the best of times it was the worst of times
it was the age of wisdom it was the age of foolishness
it was the epoch of belief it was the epoch of incredulity
it was the season of light it was the season of darkness
it was the spring of hope it was the winter of despair
```

Этот файл содержит 60 слов, из них 20 различных, а четыре из них встречаются по 10 раз (наибольшая частота). Поэтому программа FrequencyCounter может вывести любое из слов it, was, the или of (в зависимости от реализации таблицы имен), а за ним его частоту — число 10.

Понятно, что для изучения производительности для больших объемов входных данных нужны две величины. Во-первых, каждое слово из входных данных по одному разу используется в качестве ключа, поэтому, несомненно, важным является общее количество слов. И, во-вторых, каждое *отличное от предыдущих* слово заносится в таблицу имен (а общее количество таких слов равно размеру таблицы после вставки всех ключей), поэтому важно и общее количество различных слов. Для оценки времени выполнения FrequencyCounter нужно знать обе эти величины (для начала см. упражнение 3.1.6). Детали мы будем изучать при рассмотрении конкретных алгоритмов, а сейчас следует просто понять необходимость подобных приложений. Например, выполнение программы FrequencyCounter с файлом leipzig1M.txt для слов длиной не менее 8 означает миллионы поисков в таблице с сотнями тысяч ключей и значений. Веб-серверу может понадобиться обрабатывать миллиарды транзакций в таблицах с миллионами ключей и значений.

Основной вопрос, на который должны дать ответ клиент и эти примеры — можно ли разработать реализацию таблицы имен, способную выполнить огромное количество операций get() в больших таблицах, которые сами составлены в результате большого количества смеси операций get() и put()? Если нужно выполнить лишь несколько поисков, то сгодится любая реализация, но клиент вроде FrequencyCounter не справится с большими задачами без хорошей реализации таблицы имен. Программа FrequencyCounter моделирует очень распространенную ситуацию.

- Операции поиска и вставки перемешаны.
- Количество различных ключей не мало.
- Обычно нужно выполнять больше поисков, чем вставок.
- Чередование поисков и вставок непредсказуемо, но не случайно.

Нам нужно разработать реализации таблиц имен, которые позволят подобным клиентам решать типичные практические задачи.

Теперь мы рассмотрим две элементарные реализации таблиц имен и их производительность на примере клиента `FrequencyCounter`. Потом, в следующих нескольких разделах, вы познакомитесь с классическими реализациями, которые обеспечивают великолепную производительность даже для огромных входных потоков и таблиц.

Последовательный поиск в неупорядоченном связном списке

Один из примитивных вариантов структуры данных для таблицы имен — это связный список узлов, которые содержат ключи и значения, как в листинге 3.1.4. Для выполнения операции `get()` осуществляется просмотр списка, и с помощью функции `equals()` производится сравнение искомого ключа с ключами в узлах списка. Если такой ключ найден, возвращается связанное с ним значение, если нет — возвращается `null`. Для выполнения операции `put()` выполняется такой же просмотр со сравнениями искомого ключа с ключами в узлах списка. Если такой ключ найден, связанное с ним значение заменяется значением, указанным во втором аргументе, а если не найден, в начале списка создается новый узел с указанными ключом и значением (рис. 3.1.5). Этот метод называется *последовательным поиском*: поиск искомого ключа выполняется с помощью поочередного сравнения ключей с помощью функции `equals()`.

Листинг 3.1.4. Алгоритм 3.1. Последовательный поиск (в неупорядоченном связном списке)

```
public class SequentialSearchST<Key, Value>
{
   private Node first;          // Первый узел в связном списке
   private class Node
   {  // узел связного списка
      Key key;
      Value val;
      Node next;
      public Node(Key key, Value val, Node next)
      {
         this.key  = key;
         this.val  = val;
         this.next = next;
      }
   }
   public Value get(Key key)
   {  // Поиск ключа и возврат связанного с ним значения.
      for (Node x = first; x != null; x = x.next)
         if (key.equals(x.key))
            return x.val;                    // Успешный поиск
      return null;                           // Неудачный поиск
   }
   public void put(Key key, Value val)
   { // Поиск ключа. Если найден, изменяется значение, если нет — увеличивается таблица.
      for (Node x = first; x != null; x = x.next)
         if (key.equals(x.key))
         {  x.val = val; return;  }          // Найден: изменяется значение.
      first = new Node(key, val, first);     // Не найден: добавляется узел.
   }
}
```

В этой реализации ST используется закрытый внутренний класс Node — узел для хранения ключей и значений в неупорядоченном связном списке. Реализация метода get() последовательно просматривает список, чтобы узнать, содержит ли таблица указанный ключ (и если содержит, то возвращает связанное с ним значение). Операция метода put() тоже выполняет последовательный просмотр, чтобы узнать, присутствует ли ключ в таблице. Если да, то изменяется связанное с ним значение, а если нет, создается новый узел с указанными ключом и значением, который вставляется в начало списка. Реализации методов size(), keys() и энергичного delete() оставлены на самостоятельную проработку.



Рис. 3.1.5. Трассировка реализации таблицы имен связным списком для стандартного клиента индексации

Алгоритм 3.1 (SequentialSearchST) — это реализация нашего базового API таблицы имен, где используются стандартные механизмы обработки списков, которые уже применялись нами для элементарных структур данных в главе 1. Реализации методов size(), keys() и энергичного delete() оставлены на самостоятельную проработку. Рекомендуем вам выполнить эти упражнения, чтобы закрепить понимание связных списков и базового API таблицы имен.

Может ли такая реализация, основанная на связном списке, работать в приложениях, где нужны большие таблицы, вроде наших демонстрационных клиентов? Как уже было сказано, анализ алгоритмов работы с таблицами имен сложнее, чем анализ алгоритмов сортировки, из-за сложности определения последовательности операций, которые могут быть выполнены конкретным клиентом. Как было сказано в отношении программы FrequencyCounter, чаще всего бывает так, что операции поиска и вставки чередуются непредсказуемо, однако это чередование не случайно. Поэтому мы обратим пристальное внимание на производительность в худшем случае. Для краткости мы будем использовать термины *попадание* вместо "успешный поиск" и *промах* вместо "неудачный поиск".

Утверждение А. Для промахов и вставок в (неупорядоченной) таблице имен на основе связного списка с N парами ключ-значение требуется N сравнений, а для попаданий — N сравнений в худшем случае. В частности, для вставки N различных ключей в первоначально пустую таблицу на основе связного списка потребуется $\sim N^2/2$ сравнений.

Доказательство. При поиске ключа, отсутствующего в списке, каждый ключ в таблице сверяется с искомым ключом. Такой поиск приходится выполнять перед каждой вставкой — в силу политики недопустимости повторения ключей.

Следствие. Для вставки N различных ключей в первоначально пустую таблицу на основе связного списка потребуется $\sim N^2/2$ сравнений.

Правда, поиски ключей, которые *присутствуют* в таблице, не обязательно выполняются за линейное время. Одна из полезных характеристик — общее время поисков для всех ключей из таблицы, деленное на N. Эта величина в точности равна ожидаемому количеству сравнений, которое необходимо для поиска при условии, что поиски каждого ключа из таблицы выполняются с одинаковой вероятностью. Такой поиск называется *случайным попаданием*. Характер поисков в клиенте вряд ли будет полностью случайным, но он часто хорошо описывается такой моделью. Нетрудно показать, что среднее количество сравнений для случайного попадания равно $\sim N/2$: метод `get()` из алгоритма 3.1 выполняет 1 сравнение, чтобы найти первый ключ, 2 сравнения, чтобы найти второй ключ, и т.д., откуда средняя стоимость равна $(1 + 2 + \ldots + N)/N = (N + 1)/2 \sim N/2$.

Этот анализ четко показывает, что реализация связным списком с последовательным поиском работает слишком медленно, чтобы использовать ее для решения очень больших задач наподобие наших эталонных входных данных и клиентов вроде `FrequencyCounter`. Общее количество сравнений пропорционально произведению количества поисков на количество вставок — а это больше 10^9 для “Tale of Two Cities” и больше 10^{14} для “Leipzig Corpora”.

Как обычно, для проверки аналитических результатов необходимы эксперименты. В качестве примера мы исследовали работу `FrequencyCounter` с аргументом `8` и файлом `tale.txt` — это потребовало 14 350 операций `put()` (учтите, что каждое слово их входных данных приводит к вызову `put()`, чтобы изменить его счетчик, и мы игнорируем стоимость вызовов `contains()`, без которых легко обойтись).

Таблица имен разрослась до 5737 ключей, поэтому примерно треть операций увеличивала размер таблицы, а остальные две трети были поиски. Для визуального отображения производительности мы использовали объект `VisualAccumulator` (см. рис. 1.2.25), чтобы выводить две точки, соответствующие каждой операции `put()`. Для i-й операции `put()` выводятся две точки. У серой точки координата x равна i, а координата y равна количеству сравнений ключей для операции `put()`. У красной точки (на рис. 3.1.6 она черная) координата x также равна i, а координата y равна накопленному среднему числу сравнений ключей для первых i операций `put()`. Как и в случае любых научных данных, эти данные дают много информации для анализа (на нашем чертеже — по 14 350 серых и красных точек). Сейчас нас в основном интересует, подтверждает ли график нашу гипотезу, что при выполнении операции `put()` в среднем просматривается половина списка. Реальный итог немного меньше половины, но это (и точная форма кривых) больше зависит от характеристик приложения, а не от алгоритма (см. упражнение 3.1.36).

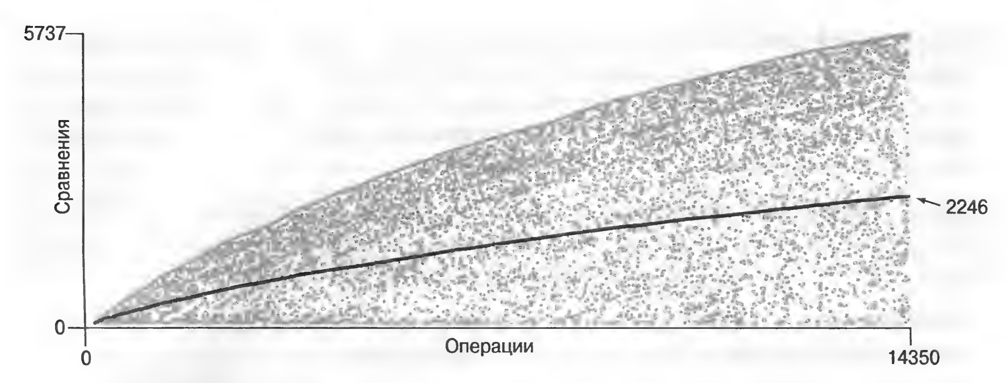


Рис. 3.1.6. Стоимость выполнения `java FrequencyCounter 8 < tale.txt` *с помощью* `SequentialSearchST`

Точное описание производительности для конкретных клиентов может оказаться сложным делом, однако отдельные гипотезы несложно формулировать и проверить с помощью программы FrequencyCount и наших эталонных входных данных или случайно упорядоченных данных — для этого удобен клиент DoublingTest, который был описан в главе 1. Мы оставим такое тестирование для самостоятельной проработки и для последующих более сложных реализаций. Если вы еще не убедились, что нам нужны более быстрые реализации, обязательно выполните эти упражнения (или просто запустите FrequencyCounter с экземпляром SequentialSearchST для файла leipzig1M.txt).

Бинарный поиск в упорядоченном массиве

Теперь мы рассмотрим полную реализацию для API упорядоченной таблицы имен. Базовой структурой данных в этом случае является пара параллельных массивов — один для ключей и один для значений. Алгоритм 3.2 (BinarySearchST в листинге 3.1.5) содержит упорядоченные ключи Comparable и использует индексацию в массивах для быстрой реализации метода get() и других операций.

Листинг 3.1.5. Алгоритм 3.2. Бинарный поиск (в упорядоченном массиве)

```
public class BinarySearchST<Key extends Comparable<Key>, Value>
{
   private Key[] keys;
   private Value[] vals;
   private int N;
   public BinarySearchST(int capacity)
   {  // Код для изменения размера массива см. в алгоритме 1.1.
      keys = (Key[]) new Comparable[capacity];
      vals = (Value[]) new Object[capacity];
   }
   public int size()
   {  return N;  }
   public Value get(Key key)
   {
      if (isEmpty()) return null;
      int i = rank(key);
      if (i < N && keys[i].compareTo(key) == 0) return vals[i];
      else                                      return null;
   }
```

```
   public int rank(Key key)
   // См. листинг 3.1.7.
   public void put(Key key, Value val)
   { // Поиск ключа. Если найден, изменяется значение; иначе таблица увеличивается.
      int i = rank(key);
      if (i < N && keys[i].compareTo(key) == 0)
      {  vals[i] = val; return;  }
      for (int j = N; j > i; j--)
      {  keys[j] = keys[j-1]; vals[j] = vals[j-1];  }
      keys[i] = key; vals[i] = val;
      N++;
   }
   public void delete(Key key)
   // См. упражнение 3.1.16.
}
```

В этой реализации ключи и значения хранятся в параллельных массивах. Реализация put() сдвигает большие ключи на одну позицию вправо, прежде чем увеличить таблицу (как в "массивной" реализации стека в разделе 1.3). Код изменения размера массивов здесь не приведен.

В основе реализации лежит метод rank() (ранг), который возвращает количество ключей, меньших заданного ключа. А в методе get() ранг точно указывает, где можно найти ключ, если он вообще есть в таблице — а если его там нет, то указывает, что он *отсутствует*.

В методе put() ранг указывает место, где нужно обновить значение, если ключ присутствует в таблице, или то место, куда нужно поместить ключ (и значение), если такого ключа в таблице не было. Все ключи, которые больше вставляемого, сдвигаются на одну позицию (начиная с задних), чтобы освободить место для вставки ключа и значения в соответствующие массивы. Как обычно, разобраться в этой структуре данных помогает изучение кода BinarySearchST вместе с трассировкой клиента тестирования (рис. 3.1.7).

		keys[]											vals[]									
Ключ	Значение	0	1	2	3	4	5	6	7	8	9	N	0	1	2	3	4	5	6	7	8	9
S	0	**S**										1	**0**									
E	1	**E**	S									2	**1**	0								
A	2	**A**	E	S								3	**2**	1	0							
R	3	A	E	**R**	S							4	2	1	**3**	0						
C	4	A	**C**	E	R	S						5	2	**4**	1	3	0					
H	5	A	C	E	**H**	R	S					6	2	4	1	**5**	3	0				
E	6	A	C	E	H	R	S					6	2	4	(6)	5	3	0				
X	7	A	C	E	H	R	S	**X**				7	2	4	6	5	3	0	**7**			
A	8	A	C	E	H	R	S	X				7	(8)	4	6	5	3	0	7			
M	9	A	C	E	H	**M**	R	S	X			8	8	4	6	5	**9**	3	0	7		
P	10	A	C	E	H	M	**P**	R	S	X		9	8	4	6	5	9	**10**	3	0	7	
L	11	A	C	E	H	**L**	M	P	R	S	X	10	8	4	6	5	**11**	9	10	3	0	7
E	12	A	C	E	H	L	M	P	R	S	X	10	8	4	(12)	5	11	9	10	3	0	7
		A	C	E	H	L	M	P	R	S	X		8	4	12	5	11	9	10	3	0	7

Рис. 3.1.7. *Трассировка реализации таблицы имен упорядоченными массивами для стандартного клиента индексации*

В алгоритме 3.2 задействованы параллельные массивы ключей и значений (альтернативный вариант см. в упражнении 3.1.12). Как и в наших реализациях обобщенных стеков и очередей из главы 1, этот код также неудобен тем, что в нем нужно создать массив `Key` типа `Comparable` и массив `Value` типа `Object`, а затем приводить их к типам `Key[]` и `Value[]`. Как обычно, можно использовать изменение размеров массивов, чтобы клиенты не заботились об этом (правда, для больших массивов этот метод работает очень медленно).

Бинарный поиск

Причина хранения ключей в упорядоченном массиве в том, что индексация массивов позволяет существенно снизить количество сравнений, необходимых для каждого поиска, с помощью классического и почтенного алгоритма *бинарного поиска*, который рассматривался в качестве примера в главе 1. При этом используются индексы, окаймляющие подмассив, в котором может находиться искомый ключ. Во время поиска этот искомый ключ сравнивается с ключом в середине подмассива. Если искомый ключ меньше ключа в середине, поиск продолжается в левой части подмассива; если искомый ключа больше ключа в середине, поиск продолжается в правой части подмассива; иначе ключ в середине равен искомому.

Код метода `rank()` в листинге 3.1.7 использует бинарный поиск для завершения рассматриваемой реализации таблицы имен. Эта реализация заслуживает внимательного изучения. Для этого мы сначала рассмотрим рекурсивный код, приведенный в листинге 3.1.6. Вызов `rank(key, 0, N-1)` выполняет ту же последовательность сравнений, что и нерекурсивная реализация в алгоритме 3.2, но в этом варианте лучше видна структура алгоритма, уже рассмотренная в разделе 1.1. Рекурсивный метод `rank()` сохраняет неизменными следующие свойства.

- Если `key` присутствует в таблице, `rank()` возвращает его индекс в таблице, и он равен количеству ключей в таблице, меньших, чем `key`.
- Если `key` отсутствует в таблице, `rank()` *также* возвращает количество ключей в таблице, меньших, чем `key`.

Листинг 3.1.6. Рекурсивный бинарный поиск

```
public int rank(Key key, int lo, int hi)
{
   if (hi < lo) return lo;

   int mid = lo + (hi - lo) / 2;
   int cmp = key.compareTo(keys[mid]);

   if      (cmp < 0)
       return rank(key, lo, mid-1);
   else if (cmp > 0)
       return rank(key, mid+1, hi);
   else return mid;
}
```

Листинг 3.1.7. Алгоритм 3.2 (продолжение). Бинарный поиск в упорядоченном массиве (итеративный)

```
public int   rank(Key key)
{
  int lo = 0, hi = N-1;
  while (lo <= hi)
  {
    int mid = lo + (hi - lo) / 2;
    int cmp = key.compareTo(keys[mid]);
    if      (cmp < 0) hi = mid - 1;
    else if (cmp > 0) lo = mid + 1;
    else return mid;
  }
  return lo;
}
```

Здесь используется классический метод (описанный в тексте) вычисления количества ключей в таблице, которые меньше key. Значение key сравнивается с ключом в середине; если они равны, возвращается его индекс; если меньше, поиск продолжается в левой половине; если больше, поиск продолжается в правой половине (рис. 3.1.8).

Успешный поиск P

lo	hi	mid	0	1	2	3	4	5	6	7	8	9
0	9	4	A	C	E	H	**L**	M	P	R	S	X
5	9	7	A	C	E	H	L	M	P	**R**	S	X
5	6	5	A	C	E	H	L	**M**	P	R	S	X
6	6	6	A	C	E	H	L	M	**P**	R	S	X

элементы a[lo..hi]

элемент a[mid]

после завершения цикла keys[mid]=P, *и возвращается* 6

Неудачный поиск Q

lo	hi	mid	0	1	2	3	4	5	6	7	8	9
0	9	4	A	C	E	H	**L**	M	P	R	S	X
5	9	7	A	C	E	H	L	M	P	**R**	S	X
5	6	5	A	C	E	H	L	**M**	P	R	S	X
7	6	6	A	C	E	H	L	M	**P**	R	S	X

после завершения цикла lo > hi, *и возвращается* 7

Рис. 3.1.8. *Трассировка бинарного поиска в упорядоченном массиве*

Время, потраченное на проверку того, что нерекурсивный вариант rank() из алгоритма 3.2 работает, как положено (либо доказав, что он эквивалентен рекурсивному варианту, либо непосредственно доказав, что цикл всегда завершается со значением lo, в точности равным количеству ключей в таблице, которые меньше key), обернется сторицей для любого программиста. (*Подсказка*: обратите внимание, что lo вначале равно 0, а потом никогда не убывает.)

Другие операции

Поскольку ключи хранятся в упорядоченном массиве, большинство операций, основанных на упорядоченности, имеют короткие и понятные реализации, как видно из листинга 3.1.8. Например, вызов `select(k)` просто возвращает `keys[k]`. Реализации методов `delete()` и `floor()` оставлены читателям в качестве самостоятельного упражнения. Мы рекомендуем вам изучить реализацию `ceiling()` и метода `keys()` с двумя аргументами и проработать упражнения, чтобы закрепить понимание API для упорядоченной таблицы имен и его реализации.

Листинг 3.1.8. Алгоритм 3.2 (продолжение). Операции в упорядоченной таблице имен

```
public Key min()
{  return keys[0];  }

public Key max()
{  return keys[N-1];  }

public Key select(int k)
{  return keys[k];  }

public Key ceiling(Key key)
{
   int i = rank(key);
   return keys[i];
}
public Key floor(Key key)
// См. упражнение 3.1.17.

public Key delete(Key key)
// См. упражнение 3.1.16.

public Iterable<Key> keys(Key lo, Key hi)
{
   Queue<Key> q = new Queue<Key>();
   for (int i = rank(lo); i < rank(hi); i++)
   q.enqueue(keys[i]);
   if (contains(hi))
      q.enqueue(keys[rank(hi)]);
   return q;
}
```

Эти методы, наряду с методами из упражнений 3.1.16 и 3.1.17, завершают реализацию нашего API для упорядоченной таблицы имен на основе бинарного поиска в упорядоченном массиве. Методы `min()`, `max()` и `select()` тривиальны: они просто возвращают ключ из известной позиции в массиве. Метод `rank()`, на котором основан бинарный поиск, является основным. Реализации `floor()` и `delete()`, оставленные на самостоятельную проработку, выглядят сложнее, но ненамного.

Анализ бинарного поиска

Рекурсивная реализация метода `rank()` приводит к заключению, что бинарный поиск гарантирует быстрое выполнение поиска, т.к. он соответствует рекурсивному соотношению, описывающему верхнюю границу количества сравнений.

Утверждение Б. Бинарный поиск в упорядоченном массиве с N ключами использует не более $\lg N + 1$ сравнений для поиска (как успешного, так и неудачного).

Доказательство. Этот анализ похож (но проще) на анализ сортировки слиянием (утверждение Е в главе 2). Пусть $C(N)$ — количество сравнений, необходимых для поиска ключа в таблице имен размером N. Очевидно, что $C(0) = 0$, $C(1) = 1$, и для $N > 0$ можно записать рекуррентное соотношение, которое непосредственно соответствует рекурсивному методу:

$$C(N) \leq C(\lfloor N/2 \rfloor) + 1$$

Когда поиск уходит влево или вправо, размер подмассива не превышает $\lfloor N/2 \rfloor$, и одно сравнение используется и для проверки на равенство, и для выбора направления продолжения поиска — слева или справа. Если N на единицу меньше степени 2 ($N = 2^n - 1$), это рекуррентное уравнение нетрудно решить. Поскольку $\lfloor \mathrm{N}/2 \rfloor = 2^{n-1} - 1$, то

$$C(2^n - 1) \leq C(2^{n-1} - 1) + 1$$

Применение такого же рассуждения к первому слагаемому в правой части дает

$$C(2^n - 1) \leq C(2^{n-2} - 1) + 1 + 1$$

Повторение этого шага еще $n-2$ раз дает

$$C(2^n - 1) \leq C(2^0) + n$$

откуда получается решение:

$$C(N) = C(2^n) \leq n + 1 < \lg N + 1$$

Точное решение для произвольного N сложнее, но сводится к применению такого же рассуждения для всех значений N (см. упражнение 3.1.20). Бинарный поиск дает гарантированное логарифмическое время поиска.

В приведенной реализации `ceiling()` метод `rank()` вызывается один раз, а в стандартной реализации `size()` — два раза, поэтому из приведенного доказательства следует, что эти операции (и `floor()`) также гарантированно выполняются за логарифмическое время (операции `min()`, `max()` и `select()` выполняются за постоянное время, см. табл. 3.1.5).

Таблица 3.1.5. Стоимости операций из `BinarySearchST`

Метод	Порядок роста времени выполнения
`put()`	N
`get()`	$\log N$
`delete()`	N
`contains()`	$\log N$
`size()`	1
`min()`	1
`max()`	1
`floor()`	$\log N$
`ceiling()`	$\log N$
`rank()`	$\log N$
`select()`	1
`deleteMin()`	N
`deleteMax()`	1

Несмотря на гарантированно логарифмическое время поиска, реализация BinarySearchST все-таки не позволяет использовать клиенты наподобие FrequencyCounter для решения очень больших задач — из-за слишком медленного метода put(). Бинарный поиск уменьшает количество сравнений, но не время выполнения, т.к. для случайно упорядоченных входных ключей количество обращений к массиву при построении таблицы имен в упорядоченном массиве квадратично относительно размера массива (в типичных практических ситуациях ключи, даже если они и не случайные, хорошо описываются данной моделью).

Утверждение Б (продолжение). Вставка нового ключа в упорядоченный массив размером N использует в худшем случае $\sim 2N$ обращений к массиву, поэтому для вставки N ключей в первоначально пустую таблицу в худшем случае требуется $\sim 2N^2$ обращений к массиву.

Доказательство. Совпадает с доказательством утверждения А.

Для "Tale of Two Cities", с более чем 10^4 различными ключами, стоимость создания таблицы равна почти 10^8 обращений к массиву, а для "Leipzig Corpora Collection", в котором более 10^6 различных ключей, стоимость создания таблицы превышает 10^{11} обращений к массиву. Это не так уж невозможно на современных компьютерах, но все-таки очень (и слишком) много.

Если снова оценить стоимость операций put() для клиента FrequencyCounter со словами длиной 8 или более, то мы увидим сокращение средней стоимости с 2246 сравнений (плюс обращения к массиву) на одну операцию для SequentialSearchST до 484 для BinarySearchST (рис. 3.1.9). Как и раньше, эта стоимость даже лучше, чем предсказанная аналитически — это, скорее всего, также объясняется свойствами приложения (см. упражнение 3.1.36). Это замечательно, но, как мы вскоре увидим, можно получить значительно более быстрые реализации.



Рис. 3.1.9. Стоимости выполнения команды java FrequencyCounter 8 < tale.txt с помощью реализации BinarySearchST

Предварительные выводы

Бинарный поиск обычно значительно лучше последовательного поиска и рекомендуется к применению во многих практических приложениях. Для статической таблицы (где не разрешены операции вставки) удобно заполнить и отсортировать таблицу, а дальше пользоваться бинарным поиском, как это было описано в главе 1 (см. рис. 1.2.26).

Даже если сначала известен большой объем пар ключ-значение, а затем выполняется много операций поиска (так часто бывает на практике), стоит добавить в класс `BinarySearchST` конструктор, который инициализирует и сортирует таблицу (см. упражнение 3.1.12). Но все-таки во многих других случаях бинарный поиск в упорядочиваемом массиве не применим. Например, он застопорится на данных "Leipzig Corpora Collection", т.к. поиски и вставки выполняются вперемешку, а размер таблицы слишком велик. Как было подчеркнуто, типичным современным клиентам поиска требуются таблицы имен, которые могут поддерживать быстрые реализации *как* поиска, *так и* вставки. То есть нужно иметь возможность создавать огромные таблицы, в которые можно вставлять (а, может, и удалять) пары ключ-значение непредсказуемого характера вперемешку с поисками.

В табл. 3.1.6 приведена сводка характеристик производительности для элементарных реализаций таблиц имен, рассмотренных в данном разделе. В этой таблице приведены лишь ведущие слагаемые стоимости (количество обращений к массиву для бинарного поиска, количество сравнений для остальных), которые в основном формируют порядок роста времени выполнения.

Таблица 3.1.6. Трудоемкости для элементарных реализаций таблицы имен

Алгоритм (структура данных)	Стоимость в худшем случае		Стоимость в среднем		Эффективно поддерживает упорядоченные операции?
	поиск	вставка	попадание	вставка	
Последовательный поиск (неупорядоченный связный список)	N	N	$N/2$	N	нет
Бинарный поиск (упорядоченный массив)	$\lg N$	$2N$	$\lg N$	N	да

Нас больше всего интересует вопрос: можно ли изобрести такие алгоритмы и структуры данных, которые позволят получить логарифмическую производительность *и* для поиска, *и* для вставки. Ответ многообещающий — да! И поиску этого ответа как раз в основном и посвящена данная глава. Наряду со способами быстрой сортировки, рассмотренными в главе 2, быстрые поиски и вставки в таблице имен представляют собой наиболее важные результаты изучения алгоритмов и наиболее важные шаги к разработке той богатой возможностями вычислительной инфраструктуры, которой мы сейчас пользуемся.

Но как достичь этой цели? Для эффективного выполнения вставок, видимо, нужна связная структура. Но односвязный список не позволяет применять бинарный поиск, т.к. эффективность бинарного поиска зависит от возможности быстро заглянуть в середину любого подмассива с помощью индексации — а единственным способом попасть в середину односвязного списка является переход по ссылкам. Для объединения эффективности бинарного поиска с гибкостью связных структур нужны более сложные структуры данных. Такое объединение обеспечивают *деревья бинарного поиска*, которые будут рассматриваться в следующих двух разделах, и *хеш-таблицы*, которым посвящен раздел 3.4.

Мы рассматриваем в данной главе шесть реализаций таблиц имен, поэтому резонно кратко ознакомиться со всеми. В табл. 3.1.7 перечислены структуры данных, а также их основные достоинства и недостатки их применения в приложениях. Они приведены в том порядке, в котором изучаются.

12 Зак. 3239

Таблица 3.1.7. Достоинства и недостатки реализаций таблиц имен

Базовая структура данных	Реализация	Достоинства	Недостатки
Связный список (последовательный поиск)	`SequentialSearchST`	Удобен для очень небольших таблиц имен	Медленно работает для больших таблиц имен
Упорядоченный массив (бинарный поиск)	`BinarySearchST`	Оптимальный поиск и объем памяти, операции, основанные на упорядоченности	Медленная вставка
Дерево бинарного поиска	`BST`	Легкость реализации, операции, основанные на упорядоченности	Нет гарантий, память для ссылок
Сбалансированное ДБП	`RedBlackBST`	Оптимальный поиск и объем памяти, операции, основанные на упорядоченности	Память для ссылок
Хеш-таблица	`SeparateChainingHashST` `LinearProbingHashST`	Быстрый поиск и вставки для обычных типов данных	Необходимо хеширование для каждого типа. Нет операций, основанных на упорядоченности. Память для ссылок или пустых элементов

По мере изучения этих алгоритмов и реализаций мы будем более подробно рассматривать их свойства, но краткие характеристики из этой таблицы помогут вам удерживать в голове общую картину. Общий вывод таков: имеются несколько быстрых реализаций таблиц имен, которые можно и нужно использовать во множестве всяких приложений.

Вопросы и ответы

Вопрос. **Почему мы не использовали для таблиц имен тип `Item`, реализующий интерфейс `Comparable` — так же, как в очередях с приоритетами в разделе 2.4 — вместо отдельных ключей и значений?**

Ответ. Можно и так. Эти два подхода демонстрируют два различных способа связывания информации с ключами: *неявный*, когда создается тип данных с ключом в нем, и *явный*, когда ключи отделены от значений. Для таблиц имен мы решили подчеркнуть абстракцию ассоциативных массивов. Кроме того, обратите внимание, что клиент указывает при поиске только ключ, а не сочетание ключ-значение.

Вопрос. **Зачем возиться с методом `equals()`? Почему не использовать везде `compareTo()`?**

Ответ. Не для всех типов данных ключи легко сравнивать, хотя работа с таблицами имен все-таки имеет смысл. Яркий пример — использование в качестве ключей графических или звуковых объектов. Нет естественного способа их сравнения, но вот проверить на равенство возможно (хотя и не очень просто).

Вопрос. **Почему ключам запрещено принимать значение `null`?**

Ответ. Предполагается, что `Key` имеет тип `Object`, т.к. он используется для вызова методов `compareTo()` или `equals()`. Но если `a` равно `null`, то вызов наподобие `a.compareTo(b)` приведет к исключению “null pointer exception”. Устранение такой возможности упрощает клиентский код.

Вопрос. **Почему не используется метод `less()`, который применялся для сортировки?**

Ответ. Равенство играет в таблицах имен особую роль, поэтому все равно нужен метод для проверки на равенство. Чтобы не плодить методы с практически одинаковой функциональностью, мы ограничиваемся встроенными в Java методами `equals()` и `compareTo()`.

Вопрос. **Почему бы в `BinarySearchST` не объявить `key[]` как `Object[]` (а не `Comparable[]`) — подобно тому, как массив `val[]` объявлен как `Object`?**

Ответ. Хороший вопрос. Если сделать так, вы получите исключение `ClassCast Exception`, поскольку ключи должны быть типа `Comparable` (это нужно для того, чтобы у `key[]` был метод `compareTo()`). Поэтому требуется объявление `key[]` как `Comparable[]`. Углубление в особенности языка программирования для точного объяснения причин уведет нас в сторону от темы главы. Мы используем эту идиому (и ничего более сложного) безо всяких изменений в любом коде, где применяются обобщенные типы и массивы `Comparable`.

Вопрос. **А что, если с одним ключом понадобится связать несколько значений? Например, как обрабатывать равные ключи при использовании в приложении типа `Date` в качестве ключа?**

Ответ. Иногда это нужно, а иногда нет. Например, на вокзале два поезда не могут одновременно прибыть на один и тот же путь (хотя могут на разные пути). В этой

ситуации возможны два выхода: либо использовать какую-то дополнительную информацию, которая снимет неоднозначность, либо использовать очередь (`Queue`) значений с одним и тем же ключом. Об этом мы поговорим подробнее в разделе 3.5.

Вопрос. **Предварительная сортировка таблицы, о которой сказано в тексте раздела — хорошая мысль. Почему ее вынесли в упражнение (см. упражнение 3.1.12)?**

Ответ. Действительно, в некоторых случаях этот способ можно рекомендовать. Однако добавление медленного метода вставки в структуру данных, предназначенную для быстрого поиска, просто "чтобы было" — это ловушка для производительности. Ведь ничего не подозревающий клиент может выполнить смесь поисков и вставок в большой таблице и захлебнуться в квадратичной сложности. Такие ловушки встречаются часто, поэтому следует быть осторожными в отношении чужого ПО со слишком широкими интерфейсами. Эта проблема становится актуальной, когда разработчик предусмотрел множество методов "для удобства" — а с ними и массу проблемных мест, если клиент ожидает, что все методы реализованы эффективно. Примером может служить класс `ArrayList`, встроенный в Java (см. упражнение 3.5.27).

Упражнения

3.1.1. Напишите клиент, который создает таблицу имен для перевода буквенных оценок в числовые (в соответствии с приведенной таблицей) и потом читает из стандартного ввода список буквенных оценок, вычисляет среднее значение этих оценок и выводит его.

A+	A	A–	B+	B	B–	C+	C	C–	D	F
4.33	4	3.67	3.33	3.00	2.67	2.33	2.00	1.67	1.00	0.00

3.1.2. Разработайте реализацию `ArrayST` для нашего API базовой таблицы имен, в которой в качестве базовой структуры данных используется (неупорядоченный) массив.

3.1.3. Разработайте реализацию `OrderedSequentialSearchST` для нашего API упорядоченной таблицы имен, в которой в качестве базовой структуры данных используется упорядоченный связный список.

3.1.4. Разработайте АТД `Time` и `Event`, которые позволяют работать с датами, как в примере на рис. 3.1.3.

3.1.5. Реализуйте методы `size()`, `delete()` и `keys()` для класса `SequentialSearchST`.

3.1.6. Приведите количество вызовов `put()` и `get()` при работе программы `FrequencyCounter` в виде функции от количества слов W и от количества различных слов D во входных данных.

3.1.7. Чему равно среднее количество различных ключей, которое может обнаружить программа `FrequencyCounter` среди N случайных неотрицательных целых чисел, меньших 1000, для $N = 10, 10^2, 10^3, 10^4, 10^5$ и 10^6?

3.1.8. Какое слово из не менее десяти букв чаще всего встречается в "Tale of Two Cities"?

3.1.9. Добавьте в программу `FrequencyCounter` код отслеживания *последнего* вызова `put()`. Выведите последнее вставленное слово и количество слов, которые были обработаны из входного потока до этой вставки. Выполните эту программу с файлом `tale.txt` для граничной длины 1, 8 и 10.

3.1.10. Приведите трассировку процесса вставки ключей `E A S Y Q U E S T I O N` в первоначально пустую таблицу с помощью реализации `SequentialSearchST`. Сколько сравнений понадобилось для этого?

3.1.11. Приведите трассировку процесса вставки ключей `E A S Y Q U E S T I O N` в первоначально пустую таблицу с помощью реализации `BinarySearchST`. Сколько сравнений понадобилось для этого?

3.1.12. Измените реализацию `BinarySearchST`, чтобы работать только с одним массивом объектов `Item`, содержащих и ключи, и значения, а не с двумя параллельными массивами. Добавьте конструктор, который принимает в качестве аргумента массив элементов `Item` и упорядочивает массив с помощью сортировки слиянием.

3.1.13. Какую из реализаций таблицы имен из представленных в данном разделе вы применили бы для приложения, которое выполняет случайную смесь из 10^3 операций `put()` и 10^6 операций `get()`? Обоснуйте свой ответ.

3.1.14. Какую из реализаций таблицы имен из представленных в данном разделе вы применили бы для приложения, которое выполняет случайную смесь из 10^6 операций `put()` и 10^3 операций `get()`? Обоснуйте свой ответ.

3.1.15. Допустим, что поиски в клиенте `BinarySearchST` выполняются в 1000 раз чаще, чем вставки. Оцените процент общего времени, которое уйдет на вставки, если количество поисков равно 10^3, 10^6 и 10^9.

3.1.16. Реализуйте метод `delete()` для класса `BinarySearchST`.

3.1.17. Реализуйте метод `floor()` для класса `BinarySearchST`.

3.1.18. Докажите, что метод `rank()` в классе `BinarySearchST` работает правильно.

3.1.19. Измените программу `FrequencyCounter` так, чтобы выводить все значения с максимальной частотой появления, а не один из них. *Совет*: воспользуйтесь классом `Queue`.

3.1.20. Завершите доказательство утверждения В (покажите, что оно верно для всех значений N). *Подсказка*: сначала покажите, что функция $C(N)$ монотонна, т.е. $C(N) \le C(N + 1)$ для всех $N > 0$.

Творческие задачи

3.1.21. *Использование памяти.* Сравните расход памяти в реализации `BinarySearchST` с расходом в `SequentialSearchST` для N пар ключ-значение при условиях, описанных в разделе 1.4. Считайте память, необходимую не для самих ключей и значений, а для ссылок на них. В `BinarySearchST` считайте, что используется изменение размера массивов, т.е. массив всегда заполнен на 25–100%.

3.1.22. *Поиск с самоорганизацией.* Алгоритм поиска с самоорганизацией переупорядочивает элементы так, чтобы те, поиск которых выполняется чаще, находились быстрее. Измените реализацию поиска из упражнения 3.1.2, чтобы при каждом

попадании найденная пара ключ-значение переносилась в начало списка, а все пары между началом списка и освободившимся местом сдвигались вправо на одну позицию. Это процедура называется эвристикой *сдвиг вперед*.

3.1.23. *Анализ бинарного поиска.* Докажите, что максимальное количество сравнений, выполняемых при бинарном поиске в таблице размером N, в точности равно количеству битов в двоичном представлении числа N, т.к. сдвиг на 1 бит вправо преобразует двоичное представление N в двоичное представление $\lfloor N/2 \rfloor$.

3.1.24. *Интерполяционный поиск.* Предположим, что с ключами можно выполнять арифметические операции (например, если это значения `Double` или `Integer`). Напишите версию бинарного поиска, которая имитирует процесс поиска в словаре: если первая буква слова находится в начале алфавита, то поиск начинается в начале словаря, и наоборот. Конкретнее, если k_x — искомое значение, k_{lo} — значение первого ключа в таблице, а k_{hi} — значение последнего ключа, то сначала нужно проверить элемент не в середине таблицы, а на относительном расстоянии $\lfloor (k_x - k_{lo}) / (k_{hi} - k_{lo}) \rfloor$ от начала. Сравните работу своей реализации с работой `BinarySearchST` для клиента `FrequencyCounter` с помощью программы `SearchCompare`.

3.1.25. *Программное кеширование.* Поскольку стандартная реализация метода `contains()` содержит вызов `get()`, во внутреннем цикле программы `FrequencyCounter`

```
if (!st.contains(word)) st.put(word, 1);
else                    st.put(word, st.get(word) + 1);
```

один и тот же ключ ищется два или три раза. Чтобы не загромождать такой код и не терять эффективность, можно использовать технику *программного кеширования*, при которой местоположение ключа, к которому было самое последнее обращение, запоминается в специальной переменной экземпляров. Измените реализации `SequentialSearchST` и `BinarySearchST` в соответствии с этой идеей.

3.1.26. *Подсчет частот из словаря.* Измените программу `FrequencyCounter`, чтобы она принимала в качестве аргумента имя файла словаря, подсчитывала частоты слов из стандартного ввода, которые присутствуют в словаре, и выводила две таблицы слов с их частотами: одна — упорядоченная по частотам, а другая — так, как в словаре.

3.1.27. *Небольшие таблицы.* Пусть клиент `BinarySearchST` выполняет S операций поиска среди N различных ключей. Приведите порядок роста S, если стоимость построения таблицы равна стоимости всех поисков.

3.1.28. *Упорядоченные вставки.* Измените реализацию `BinarySearchST` так, чтобы вставка ключа, который больше всех остальных ключей в таблице, выполнялась за постоянное время (тогда построение таблицы с помощью вызовов `put()` для упорядоченных ключей будет выполняться за линейное время).

3.1.29. *Клиент тестирования.* Напишите клиент тестирования `TestBinarySearch.java` для проверки работы реализаций методов `min()`, `max()`, `floor()`, `ceiling()`, `select()`, `rank()`, `deleteMin()`, `deleteMax()` и `keys()`, приведенных в тексте раздела. Начните со стандартного клиента индексации, приведенного в листинге 3.1.1, и при необходимости добавьте в него дополнительные аргументы командной строки.

3.1.30. *Сертификация.* Добавьте в реализацию `BinarySearchST` операторы `assert`, чтобы проверять инварианты алгоритма и целостность структуры данных после каждой вставки и удаления. Например, каждый индекс `i` должен всегда быть равен `rank(select(i))`, а массив должен быть всегда упорядочен.

Эксперименты

3.1.31. *Драйвер производительности.* Напишите программу-драйвер производительности, который заполняет таблицу имен операциями `put()`, потом использует операции `get()`, чтобы количество и попаданий, и промахов в каждый ключ таблицы было в среднем равно 10 — и все это повторяется несколько раз со случайными последовательностями строковых ключей различной длины от 2 до 50 символов. Драйвер должен измерять время каждого выполнения и выводить текстом или графически среднее время выполнения.

3.1.32. *Драйвер экстремальных проверок.* Напишите программу-драйвер проверок, которая использует методы из нашего API таблицы имен для сложных или патологических случаев, которые могут возникнуть в реальных ситуациях. Вот несколько примеров: уже упорядоченные ключи, ключи в обратном порядке, все одинаковые ключи и последовательности ключей из только двух различных значений.

3.1.33. *Драйвер для самоорганизующегося поиска.* Напишите программу-драйвер для реализации самоорганизующегося поиска (см. упражнение 3.1.22), которая заполняет таблицу имен N ключами с помощью операции `get()`, потом выполняет $10N$ успешных поисков в соответствии с заранее определенным распределением вероятности. Сравните с помощью этого драйвера время выполнения своей реализации из упражнения 3.1.22 с временем выполнения `BinarySearchST` для $N = 10^3$, 10^4, 10^5 и 10^6, используя распределение вероятности, при котором поиск i-го наименьшего ключа выполняется удачно с вероятностью $1/2^i$.

3.1.34. *Закон Зипфа.* Выполните предыдущее упражнение для распределения вероятности, при котором поиск i-го наименьшего ключа выполняется удачно с вероятностью $1/(iH_N)$, где H_N — гармоническое число (см. табл. 1.4.4). Это распределение называется *законом Зипфа* (Zipf). Сравните эвристику "сдвиг вперед" с оптимальным упорядочением для распределений из предыдущего упражнения, где ключи хранятся в порядке возрастания (по убыванию их ожидаемой частоты).

3.1.35. *Проверка производительности I.* Выполните тесты удвоения, в которых используются первые N слов из "Tale of Two Cities" для различных значений N, чтобы проверить гипотезу, что время выполнения программы `FrequencyCounter` квадратично, если в ней в качестве таблицы имен используется реализация `SequentialSearchST`.

3.1.36. *Проверка производительности II.* Объясните, почему производительность реализаций `BinarySearchST` и `SequentialSearchST` для клиента `FrequencyCounter` несколько лучше аналитического прогноза.

3.1.37. *Отношение put/get.* Эмпирически определите отношение времени, которое реализация `BinarySearchST` тратит на выполнение операций `put()` и операций `get()`, если программа `FrequencyCounter` используется для определения частот появления значений в 1 миллионе случайных M-битовых значений `int`, для $M = 10$, 20 и 30. Ответьте на этот же вопрос для файла `tale.txt` и сравните результаты.

3.1.38. *Графики амортизированной стоимости.* Доработайте реализации FrequencyCounter, SequentialSearchST и BinarySearchST, чтобы они выводили графики наподобие графиков в этом разделе для стоимости каждой операции put() во время вычисления.

3.1.39. *Замер точного времени.* Добавьте в программу FrequencyCounter использование классов Stopwatch и StdDraw для формирования графиков, где по оси X откладывается количество вызовов get() или put(), а по оси Y — общее время выполнения. Точки графика должны отображать кумулятивное время выполнения после каждого вызова. Выполните полученную программу для "Tale of Two Cities" с реализацией SequentialSearchST, а потом с реализацией BinarySearchST, и проанализируйте результаты. *Примечание*: резкие скачки на кривой объясняются *кешированием*, которое выходит за рамки этого вопроса.

3.1.40. *Исследование бинарного поиска.* Найдите значения N, для которых бинарный поиск в таблице имен размером N становится в 10, 100 и 1000 раз быстрее последовательного поиска. Предскажите значения аналитически и проверьте их экспериментально.

3.1.41. *Исследование интерполяционного поиска.* Найдите значения N, для которых интерполяционный поиск в таблице имен размером N становится в 1, 2 и 10 раз быстрее обычного бинарного поиска, если ключи — случайные 32-битовые целые числа (см. упражнение 3.1.24). Предскажите значения аналитически и проверьте их экспериментально.

3.2. Деревья бинарного поиска

В настоящем разделе мы рассмотрим реализацию таблицы имен, в которой сочетаются гибкость вставок в связный список с эффективностью поиска в упорядоченном массиве. А именно, использование *двух* ссылок на узел (вместо одной в связном списке) приводит к эффективной реализации таблицы имен, основанной на структуре данных дерева бинарного поиска, которая считается одной из наиболее фундаментальных алгоритмов в вычислительной технике.

Для начала определим нужные термины. Мы работаем со структурой данных, состоящей из *узлов*, которые содержат *ссылки*. Эти ссылки могут быть либо *нулевыми*, либо указывать на другие узлы. В *бинарном дереве* имеется ограничение: на каждый узел может указывать лишь один другой узел, который называется его *родителем* (кроме одного узла — *корня*, на который не указывает ни один узел), а каждый узел содержит ровно две ссылки, которые называются *левой* и *правой* ссылками и указывают соответственно на *левый дочерний* узел и *правый дочерний* узел. Но, хотя ссылки указывают на узлы, можно считать, что они указывают на бинарные деревья, корни которых находятся в указываемых узлах. Таким образом, бинарное дерево можно определить как либо нулевую ссылку, либо узел с левой ссылкой и правой ссылкой, каждая из которых указывает на (не пересекающиеся) *поддеревья*, которые сами являются бинарными деревьями. В *дереве бинарного поиска* каждый узел содержит также ключ и значение, на которые накладываются специальные ограничения, направленные на эффективное выполнение поиска.

Определение. *Дерево бинарного поиска (ДБП)* — это бинарное дерево, в котором каждый узел содержит ключ типа `Comparable` (и связанное с ним значение) и удовлетворяет условию, что ключ в любом узле больше ключей во всех узлах левого поддерева этого узла и меньше ключей во всех узлах правого поддерева этого узла.

На рисунках 3.2.1 и 3.2.2 мы будем помещать ключи в узлы и использовать терминологию, связывающую узлы с ключами, наподобие "`A` — левый дочерний узел `E`". Линии, соединяющие узлы, представляют ссылки, а значение, связанное с ключом, мы будем записывать черным цветом рядом с узлом (или не будем, в зависимости от контекста). Ссылки каждого узла соединяют его с узлами, расположенными ниже на рисунке, за исключением нулевых ссылок, которые обозначаются коротенькими отрезками. Как обычно, в наших примерах используются односимвольные ключи, которые генерируются нашим клиентом тестирования.

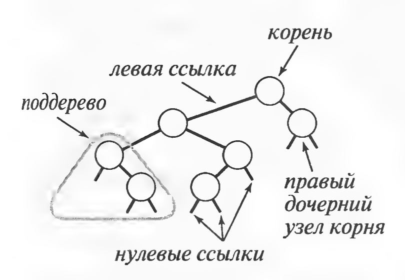


Рис. 3.2.1. Структура бинарного дерева

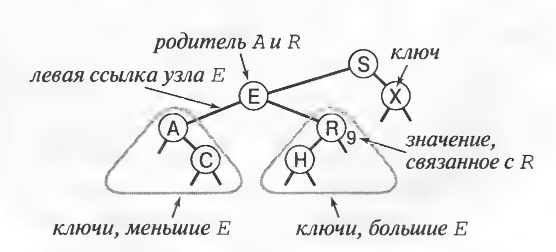


Рис. 3.2.2. Структура дерева бинарного поиска

Базовая реализация

Алгоритм 3.3 определяет структуру данных ДБП, которая будет использоваться в данном разделе для реализации API упорядоченной таблицы имен. Сначала мы рассмотрим это классическое определение структуры данных и характерные реализации методов get() (поиск) и put() (вставка).

Листинг 3.2.1. Алгоритм 3.3. Таблица имен на основе дерева бинарного поиска

```
public class BST<Key extends Comparable<Key>, Value>
{
   private Node root;                  // корень ДБП

   private class  Node
   {
      private Key key;                 // ключ
      private Value val;               // связанное значение
      private Node left, right;        // ссылки на поддеревья
      private int N;                   // к-во узлов в поддереве с этим корнем
      public Node(Key key, Value val, int N)
      {  this.key = key; this.val = val; this.N = N; }
   }

   public int size()
   {  return size(root);  }

   private int size(Node x)
   {
      if (x == null) return 0;
      else          return x.N;
   }

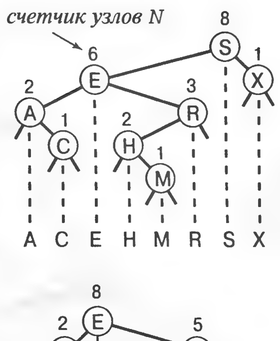
   public Value get(Key key)
   // См. листинг 3.2.2.

   public void put(Key key, Value val)
   // См. листинг 3.2.2.

   // Методы min(), max(), floor() и ceiling() см. в листинге 3.2.3.
   // Методы select() и rank() см. в листинге 3.2.4.
   // Методы delete(), deleteMin() и deleteMax() см. в листинге 3.2.5.
   // Метод keys() см. в листинге 3.2.7.
}
```

В данной реализации API упорядоченной таблицы имен используется дерево бинарного поиска, построенное из объектов Node, каждый из которых содержит ключ, связанное с ним значение, две ссылки и счетчик узлов N. Каждый такой объект является корнем поддерева, содержащего N узлов. Его левая ссылка указывает на корень Node поддерева с меньшими ключами, а правая ссылка — на корень Node поддерева с большими ключами. Переменная экземпляров root указывает на объект Node, который является корнем ДБП (содержащего все ключи и значения из таблицы имен). Реализации других методов будут приведены ниже в данном разделе.

Представление



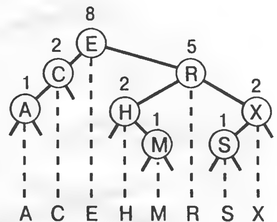


Рис. 3.2.3. Два ДБП, представляющие одно и то же множество ключей

Определение узлов в ДБП оформлено в виде приватного вложенного класса, как это было сделано для связных списков. Каждый узел содержит ключ, значение, левую ссылку, правую ссылку и счетчик узлов (на рисунках мы будем помещать эти счетчики, когда они важны, в виде цифр над узлами). Левая ссылка указывает на ДБП, узлы которого содержат меньшие ключи, а правая ссылка указывает на ДБП, узлы которого содержат бо́льшие ключи. Переменная экземпляров `N` содержит количество узлов в поддереве с корнем в данном узле. Как мы увидим, это поле облегчает реализацию различных операций в упорядоченной таблице имен. Приватный метод `size()` в алгоритме 3.3 считает, что нулевым ссылкам соответствует значение 0, так что в данном поле для каждого узла дерева `x` содержится инвариантная величина

```
size(x) = size(x.left) + size(x.right) + 1
```

ДБП представляет *набор* (множество) ключей (и связанных с ними значений), но один и тот же набор может быть представлен многими различными деревьями. Если спроецировать ключи из ДБП на горизонтальную прямую так, чтобы все ключи из левого поддерева каждого узла находились слева от ключа данного узла, а из правого поддерева — справа, то ключи всегда будут упорядоченными (рис. 3.2.3). Мы будем использовать присущую этой структуре гибкость для разработки эффективных алгоритмов построения и применения ДБП.

Поиск

Как обычно, при поиске ключа в таблице имен возможны два варианта (рис. 3.2.4). Если узел с искомым ключом присутствует в таблице, это *попадание*, и возвращается связанное с ключом значение. В противном случае это *промах*, и возвращается `null`. Рекурсивный алгоритм поиска ключа в ДБП следует непосредственно из рекурсивной структуры: если дерево пусто, это означает промах; если искомый ключ равен ключу в корне, это означает попадание. Иначе поиск (рекурсивно) продолжается в соответствующем поддереве — левом, если искомый ключ меньше, и в правом, если больше. Рекурсивный метод `get()` из листинга 3.2.2 непосредственно реализует этот алгоритм. Он принимает в качестве первого аргумента узел (корень поддерева), а в качестве второго аргумента — искомый ключ, и начинает поиск с корня всего дерева. Код поддерживает свойство, что никакие части дерева, кроме поддерева с корнем в текущем узле, не могут содержать узел с ключом, равным искомому.

Подобно тому, как размер интервала в бинарном поиске сужается на каждой итерации примерно наполовину, размер поддерева с корнем в текущем узле при поиске в ДБП уменьшается при спуске по дереву (в идеале наполовину, но не менее чем на 1). Процедура завершается, когда найден узел, содержащий искомый ключ (попадание), или когда текущее поддерево пусто (промах). Поиск начинается сверху и далее рекурсивно переходит к одному из дочерних узлов, поэтому поиск определяет путь в дереве. В случае попадания путь завершается в узле, содержащем искомый ключ, а в случае промаха путь завершается нулевой ссылкой.

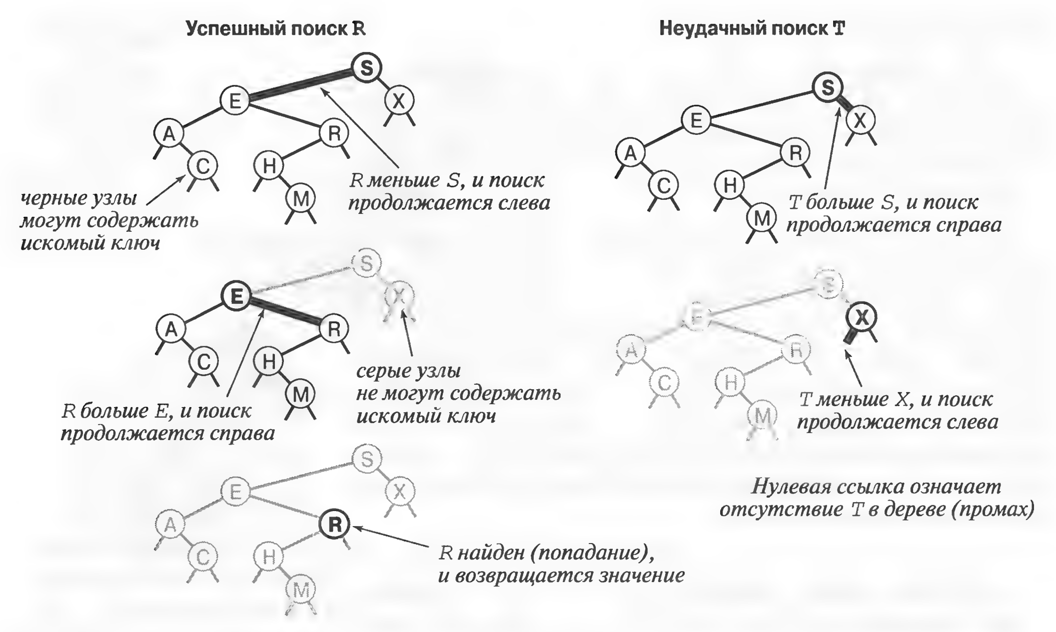


Рис. 3.2.4. Попадание (слева) и промах (справа) в ДБП

Вставка

Код поиска в алгоритме 3.3 почти так же прост, как бинарный поиск, и эта простота является важным свойством ДБП. Но более важно то, что и *вставку* в ДБП реализовать не сложнее, чем *поиск*. Ведь поиск отсутствующего ключа заканчивается на нулевой ссылке, и нужно лишь заменить эту ссылку новым узлом, содержащим вставляемый ключ (рис. 3.2.5). Рекурсивный метод put(), приведенный в алгоритме 3.3, решает эту задачу с помощью логики, похожей на рекурсивный поиск: если дерево пусто, возвращается новый узел, содержащий ключ и значение, если искомый ключ меньше ключа в корне, поиск выполняется в левом поддереве, иначе поиск выполняется в правом поддереве.

Листинг 3.2.2. Алгоритм 3.3 (продолжение). Поиск и вставка для ДБП

```
public  Value get(Key key)
{  return get(root, key);  }

private Value get(Node x, Key key)
{  // Возвращает значение, связанное с ключом в поддереве с корнем x;
   // возвращает null, если ключ в поддерреве с корнем x отсутствует.
   if (x == null) return null;
   int cmp = key.compareTo(x.key);
   if      (cmp < 0) return get(x.left, key);
   else if (cmp > 0) return get(x.right, key);
   else return x.val;
}

public void put(Key key, Value val)
{ // Поиск ключа. Если найден, изменяется значение; если нет — увеличивается дерево.
   root = put(root, key, val);
}
```

```
private Node put(Node x, Key key, Value val)
{
  // Если ключ key присутствует в поддереве с корнем x,
  // его значение заменяется на val.
  // Иначе в поддерево добавляется новый узел с ключом key и значением val.
  if (x == null) return new Node(key, val, 1);
  int cmp = key.compareTo(x.key);
  if      (cmp < 0) x.left  = put(x.left,  key, val);
  else if (cmp > 0) x.right = put(x.right, key, val);
  else              x.val = val;
  x.N = size(x.left) + size(x.right) + 1;
  return x;
}
```

Эти реализации методов get() и put() для API таблицы имен — характерные рекурсивные методы ДБП, служащие в качестве образца для нескольких других реализаций, которые будут рассмотрены ниже в данной главе. Каждый метод является и работающим кодом, и доказательством индуктивной гипотезы, сформулированной в комментарии в начале метода.

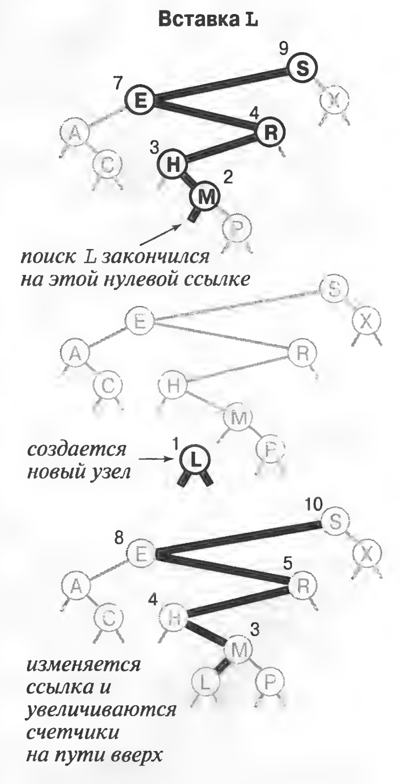


Рис. 3.2.5. Вставка в ДБП

Рекурсия

Стоит потратить немного времени, чтобы разобраться в динамике этих рекурсивных реализаций. Код *до* рекурсивных вызовов можно рассматривать как выбор пути *вниз* по дереву: он сравнивает заданный ключ с ключом в каждом узле и в соответствии с результатом сравнения переходит вправо или влево. А код *после* рекурсивных вызовов можно рассматривать как проход *вверх* по дереву. В методе get() это просто цепочка операторов возврата, а в put() такой проход выполняет изменение ссылок на дочерние узлы и счетчиков узлов при отходе по пути поиска. В простых ДБП новая ссылка образуется только внизу дерева, но перезапись ссылок на всем пути не сложнее проверки, которая позволяет отказаться от этих перезаписей. Вообще-то достаточно просто увеличивать на единицу счетчики узлов на всем пути, но мы используем более общий код, который суммирует счетчики из поддеревьев и добавляет единицу. Ниже в этом разделе и в следующем разделе мы рассмотрим более сложные алгоритмы, которые естественно выражаются этой же рекурсивной схемой, но могут изменять больше ссылок на пути поиска и требуют более общего кода вычисления счетчиков. Элементарные ДБП часто реализуются и нерекурсивным кодом (см. упражнение 3.2.12), но рекурсия в наших реализациях облегчает понимание, что код действует так, как задумано, и подготовку фундамента для более сложных алгоритмов.

Внимательное изучение трассировки нашего стандартного клиента индексации, показанной на рис. 3.2.6, поможет вам понять, как разрастаются деревья бинарного поиска.

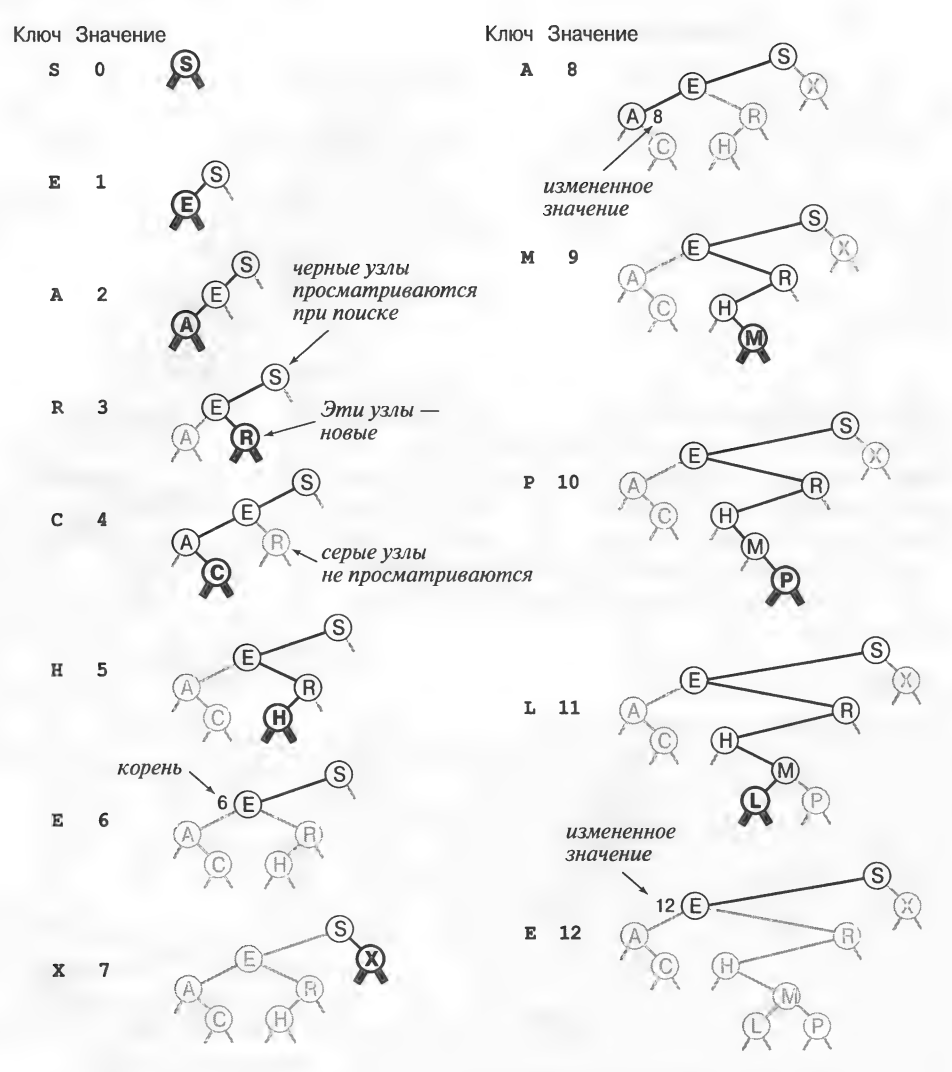


Рис. 3.2.6. Трассировка ДБП для стандартного клиента индексации

Новые узлы цепляются к нулевым ссылкам в нижней части дерева, а в остальном структура дерева не изменяется. Например, корень содержит первый вставленный ключ, один из его дочерних узлов — второй вставленный ключ и т.д. Поскольку у каждого узла две ссылки, дерево растет не только вниз, но и вширь. Более того, просматриваются только ключи на пути от корня до искомого или вставляемого ключа, поэтому по мере роста дерева доля просматриваемых ключей в дереве становится все меньше.

Анализ

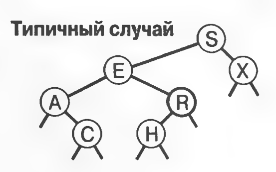
Значения времени выполнения алгоритмов на деревьях бинарного поиска зависит от формы деревьев, которая, в свою очередь, зависит от порядка, в котором заносились ключи (рис. 3.2.7). В лучшем случае дерево с N узлами должно быть полностью сбалансировано, чтобы длина пути от корня до каждой нулевой ссылки была равна $\sim\lg N$.

В худшем случае на пути поиска могут находиться все N узлов. В типичных деревьях баланс значительно ближе к лучшему, чем к худшему случаю.

Для многих случаев вполне приемлема следующая простая модель. Мы считаем, что ключи (равномерно) *случайны*, или — и это эквивалентно — что они вставляются в случайном порядке. Анализ этой модели основывается на наблюдении, что ДБП аналогичны быстрой сортировке. Узел в корне дерева соответствует первому центральному элементу в быстрой сортировке (слева нет бóльших ключей, а справа — меньших), а рекурсивное построение поддеревьев соответствует рекурсивной сортировке подмассивов. Это наблюдение приводит нас к анализу свойств деревьев.

Лучший случай

H C S A E R X



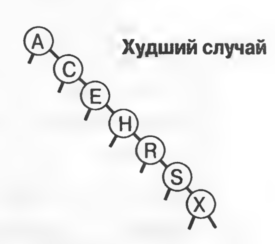


Рис. 3.2.7. Различные варианты ДБП

Утверждение В. Для попаданий в ДБП, построенном из N случайных ключей, в среднем требуется $\sim 2\ln N$ (примерно $1{,}39\lg N$) сравнений.

Доказательство. Количество сравнений, которые выполняются при успешном поиске некоторого узла, равно 1 плюс глубина этого узла. Сложив глубины всех узлов, мы получим величину, которая называется *длиной внутреннего пути* этого дерева. А нужная нам величина равна 1 плюс средняя длина внутреннего пути ДБП, которую можно проанализировать так же, как и в утверждении Б в разделе 2.3. Пусть C_N — полная длина внутреннего пути ДБП, построенного вставками N случайно упорядоченных различных ключей; тогда средняя стоимость попадания равна $1 + C_N/N$. Очевидно, что $C_0 = C_1 = 0$, а для $N > 1$ можно записать рекуррентное соотношение, которое непосредственно отражает рекурсивную структуру деревьев:

$$C_N = N-1 + (C_0 + C_{N-1}) / N + (C_1 + C_{N-2}) / N + \ldots + (C_{N-1} + C_0) / N$$

Слагаемое $N-1$ учитывает добавку единицы в длину пути для каждого из остальных $N-1$ узлов дерева; остальная часть выражения соответствует поддеревьям, которые могут с одинаковой вероятностью иметь любой из N размеров. После переупорядочивания слагаемых полученное рекуррентное соотношение почти идентично тому, которое мы уже решили в разделе 2.3 для быстрой сортировки, и из него следует, что $C_N \sim 2N\ln N$.

Утверждение Г. Для вставок и промахов в ДБП, построенном из N случайных ключей, в среднем требуется $\sim 2\ln N$ (примерно $1{,}39\lg N$) сравнений.

Доказательство. При вставках и промахах по сравнению с попаданиями в среднем выполняется одно дополнительное сравнение. Этот факт нетрудно обосновать методом индукции (см. упражнение 3.2.16).

Утверждение В гласит, что стоимость поиска в ДБП для случайных ключей примерно на 39% выше, чем для бинарного поиска. Но из утверждения Г следует, что эти затраты оправданны, т.к. стоимость вставки нового ключа также логарифмична — гибкость, не доступная бинарному поиску в упорядоченном массиве, где количество обращений к массиву при вставке элементов обычно линейно.

Эксперименты

Насколько хорошо наша модель со случайными ключами соответствует типичным ситуациям, с которыми сталкиваются клиенты таблиц имен (см. например, рис. 3.2.8)? Конечно, этот вопрос следует тщательно изучить для конкретных практических приложений, поскольку возможны большие различия в производительности. К счастью, для многих клиентов эта модель вполне адекватно описывает ДБП.

В нашем примере исследования стоимости операций `put()` в программе `FrequencyCounter` для слов длиной не менее 8 наблюдается снижение средней стоимости с 484 обращений к массиву или сравнений на одну операцию для класса `BinarySearchST` до 13 для `BST` — еще одно небольшое подтверждение логарифмической производительности, предсказанной теоретической моделью (рис. 3.2.9). Результаты других обширных экспериментов для более объемных входных данных приведены в табл. 3.2.1. На основе утверждений В и Г можно предсказать, что это количество будет примерно равно двум логарифмам размера таблицы, т.к. в основном будут выполняться поиски в почти заполненной таблице. Этот прогноз грешит, по крайней мере, следующими врожденными неточностями.

- Многие операции выполняются в небольших таблицах.
- Ключи не случайны.
- Размер таблицы может быть слишком мал для точной аппроксимации $2\ln N$.

Таблица 3.2.1. Среднее количество сравнений на одну операцию `put()` для клиента `FrequencyCounter` с `BST`

	`tale.txt`				`leipzig1M.txt`			
	Слова	Различные	Сравнения		Слова	Различные	Сравнения	
			Прогноз	Реально			Прогноз	Реально
Все слова	135 635	10 679	18.6	17.5	21 191 455	534 580	23.4	22.1
Слова ≥ 8	14 350	5 737	17.6	13.9	4 239 597	299 593	22.7	21.4
Слова ≥ 10	4 582	2 260	15.4	13.1	1 610 829	165 555	20.5	19.3

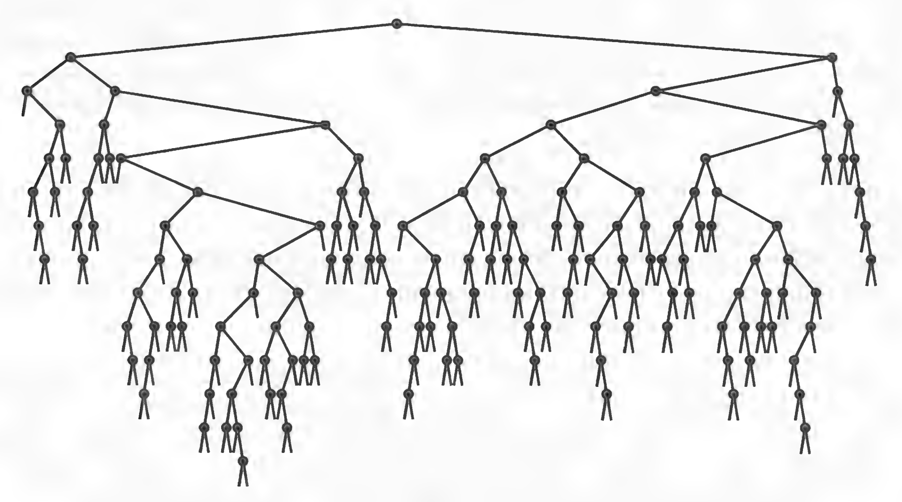


Рис. 3.2.8. Типичное ДБП, построенное из 256 случайных ключей

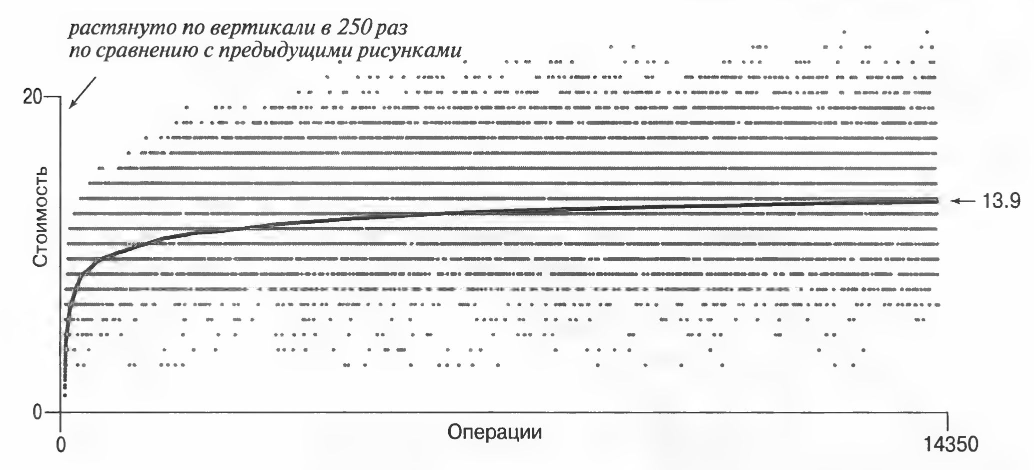


Рис. 3.2.9. Стоимости выполнения команды `java FrequencyCounter 8 < tale.txt` *с помощью* `BST`

Однако, как видно из табл. 3.2.1, для клиента `FrequencyCounter` приведенный прогноз все-таки совпадает с замерами с точностью до нескольких сравнений. Эти различия в основном можно объяснить уточнением математических выкладок (см. упражнение 3.2.35).

Методы, основанные на упорядоченности, и удаление

Важной причиной широкого применения ДБП является их способность *поддерживать упорядоченность ключей*. Поэтому на их основе можно реализовать много методов из нашего API упорядоченной таблицы имен (см. рис. 3.1.2), которые позволяют клиентам обращаться к парам ключ-значение не только по ключу, но и по относительному положению ключа. Сейчас мы рассмотрим реализации различных методов их этого API.

Минимум и максимум

Если левая ссылка корневого узла нулевая, то наименьшим ключом в ДБП является ключ в корне, а если она отлична от `null`, то наименьшим ключом в ДБП является наименьший ключ в поддереве с корнем, на который указывает левая ссылка. Данное предложение представляет собой и описание рекурсивного метода `min()` из листинга 3.2.3, и индуктивное доказательство, что он находит в ДБП наименьший ключ. Это вычисление эквивалентно простой итерации (переходы вниз-влево до нулевой ссылки), но для однотипности мы используем рекурсию. Можно было бы написать метод, возвращающий только ключ, а не весь узел, но позже нам понадобится метод для обращения к узлу, содержащему минимальный ключ. Максимальный ключ находится аналогично, только переходы выполняются вправо.

Листинг 3.2.3. Алгоритм 3.3 (продолжение). Функции `min()`, `max()`, `floor()` и `ceiling()` в ДБП

```
public Key min()
{
   return min(root).key;
}
```

```
private Node min(Node x)
{
   if (x.left == null) return x;
   return min(x.left);
}
public Key floor(Key key)
{
   Node x = floor(root, key);
   if (x == null) return null;
   return x.key;
}
private Node floor(Node x, Key key)
{
   if (x == null) return null;
   int cmp = key.compareTo(x.key);
   if (cmp == 0) return x;
   if (cmp < 0)  return floor(x.left, key);
   Node t = floor(x.right, key);
   if (t != null) return t;
   else           return x;
}
```

Каждый клиентский метод вызывает соответствующий приватный метод, который принимает в качестве аргумента дополнительную ссылку (на `Node`) и возвращает `null` или узел, содержащий запрошенный ключ, с помощью рекурсивной процедуры, описанной в тексте. Методы `max()` и `ceiling()` совпадают с `min()` и `floor()`, за исключением замены направления влево направлением вправо (и < на >).

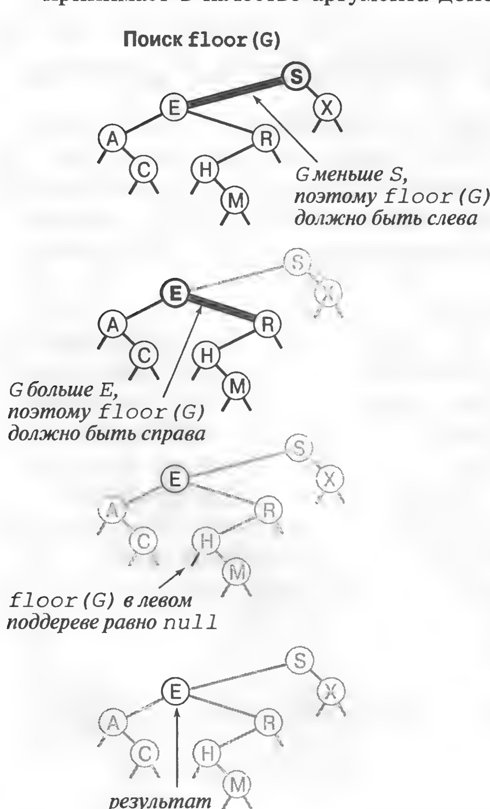


Рис. 3.2.10. Вычисление функции `floor()`

Нижняя и верхняя опоры

Если данный ключ *меньше* ключа в корне ДБП, то нижняя опора ключа (наибольший ключ в ДБП, меньший или равный заданному) *должна* находиться в левом поддереве. А если он *больше* ключа в корне, то нижняя опора ключа *может* находиться в правом поддереве, но только если в правом поддереве имеется ключ, меньший или равный заданному ключу. Если это не так (или если данный ключ равен ключу в корне), то нижней опорой данного ключа является ключ в корне. Это описание (и рис. 3.2.10) также служит и основой для рекурсивного метода `floor()`, и индуктивным доказательством, что он вычисляет то, что нужно. Замена направления вправо направлением влево (и *меньше* на *больше*) даст описание метода `ceiling()`.

Выбор

Принцип выбора в ДБП похож на метод выбора в массиве, основанный на разбиении, который мы рассматривали в разделе 2.5. В узлах ДБП имеется переменная N, содержащая количество ключей в поддереве с корнем в этом узле — она как раз поддерживает данную операцию.

Допустим, нам нужно найти ключ ранга k (такой ключ, что в точности k других ключей в дереве меньше его). Если количество ключей t в левом поддереве больше k, мы ищем (рекурсивно) ключ ранга k в левом поддереве, если t равно k, возвращаем ключ из корневого узла, а если t меньше k, мы ищем (рекурсивно) ключ ранга $k-t-1$ в правом поддереве. Как обычно, это описание служит и основой для рекурсивного метода select() из листинга 3.2.4, и доказательством по индукции, что этот метод работает так, как надо.

Листинг 3.2.4. Алгоритм 3.3 (продолжение). Выбор и ранг в ДБП

```
public Key select(int k)
{
  return select(root, k).key;
}

private Node select(Node x, int k)
{  // Возвращает узел, содержащий ключ ранга k.
   if (x == null) return null;
   int t = size(x.left);
   if      (t > k) return select(x.left,  k);
   else if (t < k) return select(x.right, k-t-1);
   else            return x;
}

public int rank(Key key)
{  return rank(key, root);  }

private int rank(Key key, Node x)
{  // Возвращает количество ключей, меньших x.key, в поддереве с корнем в x.
  if (x == null) return 0;
  int cmp = key.compareTo(x.key);
  if      (cmp < 0) return rank(key, x.left);
  else if (cmp > 0) return 1 + size(x.left) + rank(key, x.right);
  else              return size(x.left);
}
```

В этом коде та же рекурсивная схема, что и во всей главе, используется для реализации методов select() и rank(). Здесь задействован приватный метод size(), который приведен в начале данного раздела — он возвращает количество поддеревьев с корнями в каждом узле.

Ранг

Обратный метод rank(), возвращающий ранг заданного ключа, работает похожим образом (рис. 3.2.11): если заданный ключ равен ключу в корне, возвращается количество ключей t в левом поддереве; если заданный ключ меньше ключа в корне, возвращается ранг ключа в левом поддереве (вычисленный рекурсивно); а если заданный ключ больше ключа в корне, возвращается t плюс 1 (чтобы учесть ключ в корне) плюс ранг ключа в правом поддереве (вычисленный рекурсивно).

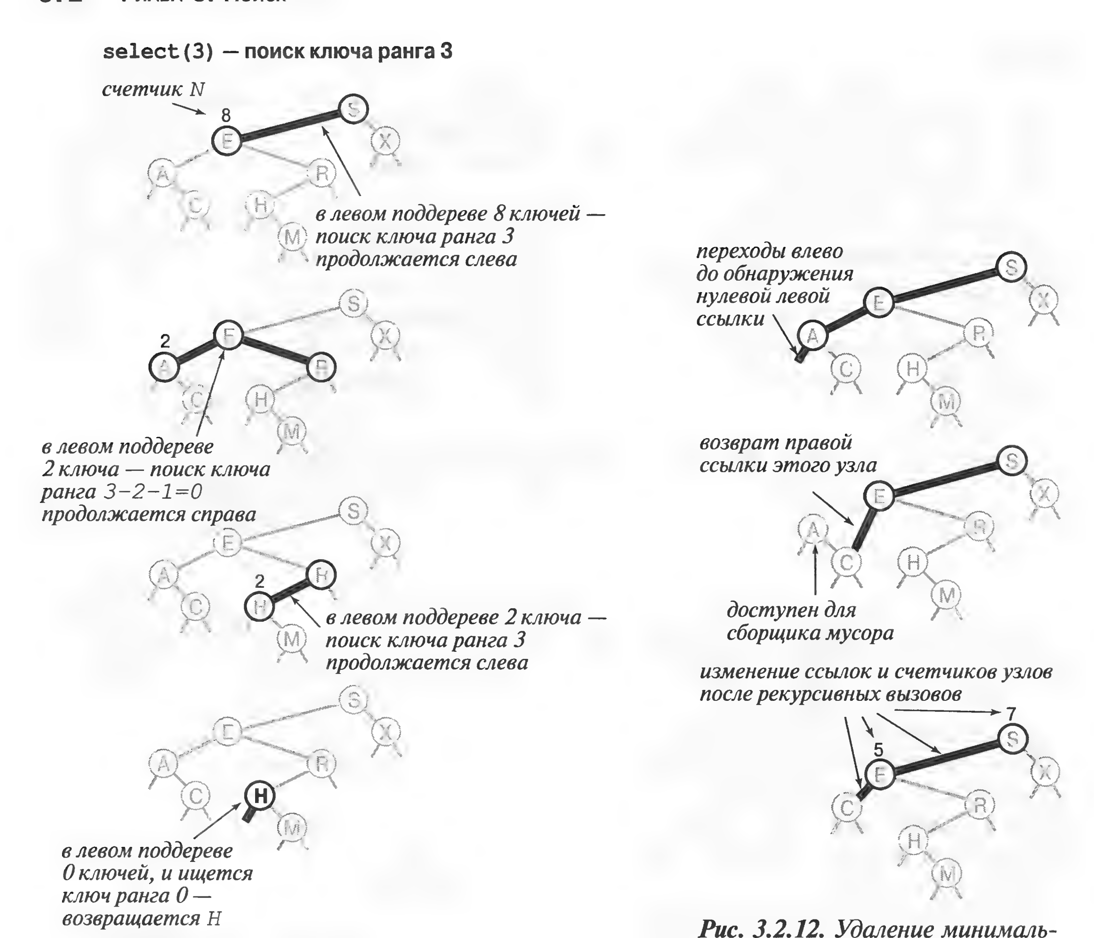


Рис. 3.2.11. Выбор в ДБП

Рис. 3.2.12. Удаление минимального ключа в ДБП

Удаление минимума/максимума

Сложнее всего в ДБП реализовать метод `delete()`, который удаляет из таблицы имен пару ключ-значение. Для начала рассмотрим более простой метод `deleteMin()` — удаление наименьшего ключа и соответствующего ему значения (рис. 3.2.12). Как и метод `put()`, это рекурсивный метод, который принимает в качестве аргумента ссылку на узел `Node` и возвращает ссылку на `Node`, отражающую результат изменений в дереве. В методе `deleteMin()` мы проходим по левой стороне, пока не встретится узел с нулевой левой ссылкой, и заменяем ссылку на этот узел его правой ссылкой (просто возвратив правую ссылку в рекурсивном методе). После этого на удаленный узел уже не указывает никакая ссылка, и он доступен сборщику мусора. Затем рекурсивная процедура завершает свою задачу, изменив соответствующую ссылку в родительском узле и счетчики во всех узлах на пути до корня. Симметричный метод работает для удаления максимума.

Удаление

Аналогичным образом можно удалить любой узел с одним дочерним узлом (или вовсе без них), но как удалить узел с двумя дочерними узлами? У нас две ссылки, но в роди-

тельском узле место только для одной. Ответ на эту дилемму впервые дал Т. Хиббард (T. Hibbard) в 1962 г.: он предложил для удаления узла x заменить этот узел его *преемником*. Поскольку у x имеется правый дочерний узел, его преемник — это узел с наименьшим ключом в правом поддереве. Такая замена сохранит упорядоченность дерева, т.к. между x.key и ключом преемника нет других ключей. Задачу замены x его преемником можно выполнить за четыре (!) легких шага (рис. 3.2.13).

- Сохраняем ссылку на удаляемый узел в t.
- Заносим в x ссылку на его преемника min(t.right).
- В правую ссылку x (которая должна указывать на ДБП, содержащее все ключи, большие x.key) заносим deleteMin(t.right) — ссылку на ДБП, содержащее все ключи, большие x.key, после удаления.
- В левую ссылку x (которая была нулевой) заносим t.left (все ключи, меньшие и удаленного ключа, и его преемника).

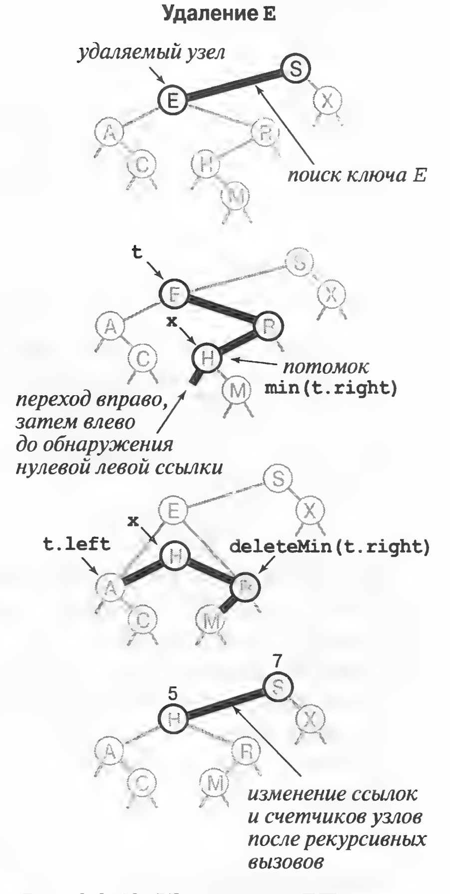


Рис. 3.2.13. Удаление в ДБП

После рекурсивных вызовов наша стандартная рекурсивная схема выполняет задачу изменения соответствующей ссылки в родительском узле и уменьшения счетчиков узлов в узлах на пути к корню (здесь также эти счетчики вычисляются как сумма счетчиков в его дочерних узлах плюс один). Этот метод действительно делает то, что надо, но у него есть недостаток, который может привести к проблемам с производительностью в некоторых реальных ситуациях. Проблема эта в том, что выбор преемника произволен и не симметричен. Почему не использовать предшественника? На практике лучше выбирать случайным образом или предшественника, или преемника. Подробнее см. в упражнении 3.2.42.

Листинг 3.2.5. Алгоритм 3.3 (продолжение). Удаление в ДБП

```
public void deleteMin()
{
  root = deleteMin(root);
}

private Node deleteMin(Node x)
{
  if (x.left == null) return x.right;
  x.left = deleteMin(x.left);
  x.N = size(x.left) + size(x.right) + 1;
  return x;
}
```

```
public void delete(Key key)
{  root = delete(root, key);  }

private Node delete(Node x, Key key)
{
   if (x == null) return null;
   int cmp = key.compareTo(x.key);
   if      (cmp < 0) x.left  = delete(x.left,  key);
   else if (cmp > 0) x.right = delete(x.right, key);
   else
   {
      if (x.right == null) return x.left;
      if (x.left == null) return x.right;
      Node t = x;
      x = min(t.right);  // См. листинг 3.2.3.
      x.right = deleteMin(t.right);
      x.left = t.left;
   }
   x.N = size(x.left) + size(x.right) + 1;
   return x;
}
```

Этот метод реализует энергичное удаление Хиббарда в ДБП, описанное в тексте. Код delete() компактен, но понять его непросто. Пожалуй, лучше всего прочитать его описание в тексте, попробовать написать код самостоятельно на основе описания, а затем сравнить то, что получится, с данным кодом. Приведенный здесь метод обычно вполне эффективен, но в крупных приложениях производительность может оказаться под вопросом (см. упражнение 3.2.42). Метод deleteMax() почти совпадает с deleteMin(), с точностью до смены направлений вправо и влево.

Запросы диапазонов

Поиск в диапазоне [F..T]

ключи, выделенные жирным, используются в сравнениях, но не попадают в диапазон

такие ключи попадают в диапазон

***Рис. 3.2.14.** Поиск в диапазоне в ДБП*

Чтобы реализовать метод keys(), который возвращает ключи в заданном диапазоне, мы начнем с простого рекурсивного метода обхода ДБП, который называется *поперечным* (inorder) *обходом*. Для демонстрации этого метода мы рассмотрим задачу вывода всех ключей в ДБП в упорядоченном виде. Для этого нужно вывести все ключи из левого поддерева (которые меньше ключа в корне по определению ДБП), потом вывести ключ в корне, а потом все ключи из правого поддерева (которые больше ключа в корне по определению ДБП) — как показано на рис. 3.2.14 и в листинге 3.2.6. Как обычно, это описание заодно служит и доказательством методом индукции, что этот код выводит ключи именно по порядку.

Чтобы реализовать метод keys() с двумя аргументами, который возвращает клиенту все ключи в заданном диапазоне, мы дополнили его добавлением в очередь Queue каждого ключа, который принадлежит диапазону, и пропуском рекурсивных вызовов поддеревьев, которые не могут содержать ключи из этого диапазона. Как и в классе BinarySearchST, накопление ключей в очереди скрыто от клиента. Это сделано для

того, чтобы клиенты могли обрабатывать все ключи из нужного им диапазона с помощью конструкции *foreach*, не зная, какая структура данных используется для реализации типа Iterable<Key>.

Листинг 3.2.6. Вывод ключей из ДБП в упорядоченном виде

```
private void print(Node x)
{
  if (x == null) return;
  print(x.left);
  StdOut.println(x.key);
  print(x.right);
}
```

Листинг 3.2.7. Алгоритм 3.3 (продолжение). Поиск диапазона в ДБП

```
public Iterable<Key> keys()
{  return keys(min(), max());  }

public Iterable<Key> keys(Key lo, Key hi)
{
   Queue<Key> queue = new Queue<Key>();
   keys(root, queue, lo, hi);
   return queue;
}

private void keys(Node x, Queue<Key> queue, Key lo, Key hi)
{
  if (x == null) return;
  int cmplo = lo.compareTo(x.key);
  int cmphi = hi.compareTo(x.key);
  if (cmplo < 0) keys(x.left, queue, lo, hi);
  if (cmplo <= 0 && cmphi >= 0) queue.enqueue(x.key);
  if (cmphi > 0) keys(x.right, queue, lo, hi);
}
```

Чтобы занести в очередь все ключи из дерева с корнем в заданном узле, которые попадают в указанный диапазон, мы (рекурсивно) добавляем в очередь все ключи из левого поддерева (если там есть такие), потом ключ из корневого узла (если он попадает в диапазон), и затем (рекурсивно) добавляем все ключи из правого поддерева (если там есть такие).

Анализ

Насколько эффективны операции в ДБП, основанные на их упорядоченности? Для ответа на этот вопрос мы рассмотрим *высоту дерева* — максимальную глубину любого узла в дереве. Для конкретного дерева его высота определяет стоимость всех операций ДБП в худшем случае (кроме поиска диапазона, где добавляется стоимость, пропорциональная количеству возвращаемых ключей).

Утверждение Д. Все операции в ДБП в худшем случае выполняются за время, пропорциональное длине дерева.

Доказательство. Все эти методы проходят вниз по дереву по одному или двум путям. По определению, длина любого такого пути не превышает высоту дерева.

Мы ожидаем, что высота дерева (стоимость в худшем случае) больше средней длины внутреннего пути (которая усредняется и с короткими путями) — но насколько больше? С виду этот вопрос похож на вопросы, на которые даны ответы в утверждениях В и Г, но на самом деле ответить на него гораздо сложнее — настолько сложнее, что мы не будем отвечать на него в данной книге. Дж. Робсон (J. Robson) в 1979 г. показал, что средняя высота ДБП логарифмична, а позже Л. Девро (L. Devroye) показал, что для больших N это значение приблизительно равно $2{,}99\lg N$. Так что если вставки в приложении хорошо описываются моделью со случайными ключами, то мы неплохо продвинулись в разработке реализации таблицы имен, которая выполняет все эти операции за логарифмическое время. Можно ожидать, что в дереве, построенном из случайных ключей, ни один путь не будет длиннее $3\lg N$, но чего следует ожидать, если ключи не случайны? В следующем разделе вы узнаете, что этот вопрос не важен на практике — из-за *сбалансированных ДБП*, которые гарантируют логарифмическую высоту дерева независимо от порядка вставки ключей.

Итак, деревья бинарного поиска нетрудно реализовать, и они могут обеспечить быстрый поиск и вставку для разнообразных практических приложений — *если* вставляемые ключи хорошо согласуются с моделью случайных ключей. Для наших примеров (и многих реальных приложений) ДБП означают разницу между возможностью и невозможностью решить какую-то задачу. Кроме того, многие программисты выбирают реализацию таблиц имен с помощью ДБП потому, что они позволяют быстро выполнять операции ранга, выбора, удаление и запроса диапазона (табл. 3.2.2). Однако, как было сказано, плохая производительность ДБП в худшем случае может оказаться неприемлемой в некоторых ситуациях. Высокая производительность базовой реализации ДБП зависит от малой вероятности наличия длинных путей в дереве, а это зависит от того, насколько вставляемые ключи похожи на случайные.

В случае быстрой сортировки можно было выполнить предварительное случайное перемешивание, но в API таблиц имен такой возможности нет, т.к. вид смеси операций зависит от клиента. Вообще-то худший случай не так уж маловероятен на практике: он возникает тогда, когда клиент вставляет ключи в прямом или обратном порядке, а такая последовательность вполне может быть выполнена безо всяких предупреждений. Эта возможность и является основной причиной поиска лучших алгоритмов и структур данных, которые будут рассмотрены в последующих разделах.

Таблица 3.2.2. Трудоемкости для элементарных реализаций таблицы имен (обновленная)

Алгоритм (структура данных)	Стоимость в худшем случае (после *N* вставок)		Средняя стоимость (после *N* случайных вставок)		Эффективно поддерживает упорядоченные операции?
	поиск	вставка	попадание	вставка	
Последовательный поиск (неупорядоченный связный список)	N	N	$N/2$	N	нет
Бинарный поиск (упорядоченный массив)	$\lg N$	N	$\lg N$	$N/2$	да
Дерево бинарного поиска (ДБП)	N	N	$1{,}39\lg N$	$1{,}39\lg N$	да

Вопросы и ответы

Вопрос. **Раньше встречались реализации ДБП, но без рекурсии. Чем они лучше или хуже?**

Ответ. Обычно рекурсивные реализации проще проверить на корректность, а нерекурсивные реализации работают немного эффективнее. Реализация метода `get()` из упражнения 3.2.13 — один из примеров, когда повышение эффективности заметно. В рекурсивной реализации в случае несбалансированных деревьев глубина стека вызовов может стать проблемой. Основная причина использования рекурсии в этой книге — облегчение перехода к реализациям сбалансированных ДБП в следующем разделе, которые легче реализовать и отлаживать в рекурсивном варианте.

Вопрос. **Поддержка счетчика узлов в поле `Node` требует объемного кода. Неужели это так необходимо? Почему бы не использовать единственную переменную экземпляров, содержащую количество узлов во всем дереве, которое необходимо для клиентского метода `size()`?**

Ответ. Методам `rank()` и `select()` нужны размеры поддеревьев с корнями в каждом узле. Если эти операции, основанные на упорядоченности, вам не нужны, можно упростить код, убрав это поле (см. упражнение 3.2.12). Конечно, хранение счетчиков во всех узлах может привести к ошибкам, но может послужить и хорошей проверкой при отладке. Реализацию метода `size()` для клиентов можно выполнить в виде рекурсивного метода, но для подсчета всех узлов ей понадобится *линейное* время, а это чревато снижением производительности клиентской программы, если ее автор не знает, что такая простая операция может оказаться такой трудоемкой.

Упражнения

3.2.1. Нарисуйте ДБП, полученное при вставке ключей `E A S Y Q U E S T I O N` в указанном порядке (присваивая `i`-у ключу значение `i`, как это принято в тексте) в первоначально пустое дерево. Сколько сравнений понадобится для построения этого дерева?

3.2.2. Вставка ключей в порядке `A X C S E R H` в первоначально пустое ДБП приводит к худшему случаю, когда каждый узел дерева имеет одну нулевую ссылку (кроме самого нижнего, у которого две нулевых ссылки). Приведите пять других перестановок этих ключей, которые также приведут к появлению деревьев худшего случая.

3.2.3. Приведите пять перестановок ключей `A X C S E R H`, которые при вставке в первоначально пустое ДБП приведут к появлению деревьев *лучшего случая*.

3.2.4. Пусть некоторое ДБП содержит целочисленные ключи от `1` до `10`, и в нем ищется ключ `5`. Которая из приведенных ниже последовательностей *не может* быть последовательностью просматриваемых при поиске ключей?

а) `10, 9, 8, 7, 6, 5`
б) `4, 10, 8, 7, 53`
в) `1, 10, 2, 9, 3, 8, 4, 7, 6, 5`
г) `2, 7, 3, 8, 4, 5`
д) `1, 2, 10, 4, 8, 5`

3.2.5. Предположим, что заранее имеется оценка частот, с которыми будет выполняться обращение к каждому ключу ДБП. Вставлять эти ключи в дерево можно в произвольном порядке. Как лучше вставлять ключи — в порядке возрастания частоты обращений, в порядке убывания или в каком-то другом порядке? Обоснуйте свой ответ.

3.2.6. Добавьте в класс `BST` метод `height()` для вычисления высоты дерева. Разработайте две реализации: рекурсивный метод (требующий линейного времени и памяти, пропорциональной высоте) и метод наподобие `size()`, для работы которого нужно дополнительное поле в каждом узле дерева (требующий линейного объема памяти и константного времени на запрос).

3.2.7. Добавьте в класс `BST` рекурсивный метод `avgCompares()` для вычисления среднего количества сравнений, необходимых для случайного попадания в заданном ДБП (длина внутреннего пути дерева, деленная на его размер, плюс один). Разработайте две реализации: рекурсивный метод (требующий линейного времени и объема памяти, пропорционального высоте дерева) и метод вроде `size()`, для работы которого нужно дополнительное поле в каждом узле дерева (требующий линейного объема памяти и константного времени на запрос).

3.2.8. Напишите статический метод `optCompares()`, который принимает целочисленный аргумент `N` и вычисляет количество сравнений, необходимых для случайного попадания в оптимальном (идеально сбалансированном) ДБП, где все нулевые ссылки находятся на одном и том же уровне, если количество ссылок равно степени 2, или иначе на двух соседних уровнях.

3.2.9. Начертите все различные формы ДБП, которые могут получиться после вставки N ключей в первоначально пустое дерево, для $N = 2$, 3, 4, 5 и 6.

3.2.10. Напишите клиент тестирования `TestBST.java` для проверки работы приведенных в тексте реализаций методов `min()`, `max()`, `floor()`, `ceiling()`, `select()`, `rank()`, `delete()`, `deleteMin()`, `deleteMax()` и `keys()`. Начните со стандартного клиента индексации, приведенного в листинге 3.1.2. Добавьте в него код, необходимый для приема дополнительных аргументов из командной строки.

3.2.11. Сколько возможно форм двоичных деревьев из N узлов и высотой N? Сколько существует различных способов вставки N различных ключей в первоначально пустое ДБП, которые приводят к появлению дерева высотой N? (См. упражнение 3.2.2.)

3.2.12. Разработайте реализацию API для ДБП, в которой нет методов `rank()` и `select()`, а в узлах нет поля счетчика.

3.2.13. Приведите нерекурсивные реализации методов `get()` и `put()` для класса `BST`.

Частичное решение. Вот реализация метода `get()`:

```
public Value get(Key key)
{
   Node x = root;
   while (x != null)
   {
      int cmp = key.compareTo(x.key);
      if (cmp == 0) return x.val;
      else if (cmp < 0) x = x.left;
      else if (cmp > 0) x = x.right;
   }
   return null;
}
```

Реализация метода `put()` не так проста, поскольку нужно сохранять указатель на родительский узел, чтобы связать его с новым узлом внизу. Кроме того, из-за необходимости изменения счетчиков требуется дополнительный проход для проверки, присутствует ли уже ключ в таблице. Поскольку в реализациях, критичных для производительности, выполняется значительно больше поисков, чем вставок, применение этого кода для метода `get()` оправдана; соответствующее изменение для метода `put()` может остаться незамеченным.

3.2.14. Приведите нерекурсивные реализации методов `min()`, `max()`, `floor()`, `ceiling()`, `rank()` и `select()`.

3.2.15. Приведите последовательность узлов, просмотренных методами класса `BST` при вычислении каждого из следующих выражений для дерева на рис. 3.2.15.

а) `floor("Q")`

б) `select(5)`

в) `ceiling("Q")`

г) `rank("J")`

д) `size("D", "T")`

е) `keys("D", "T")`

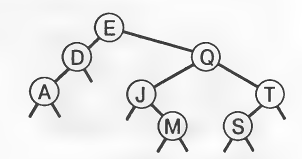


Рис. 3.2.15. *Дерево для упражнения 3.2.15*

3.2.16. Определим *длину внешнего пути* дерева как сумму количества узлов на пути от корня до всех нулевых ссылок. Докажите, что разность между длинами внешнего и внутреннего путей в любом бинарном дереве с N узлами равна $2N$ (см. утверждение В).

3.2.17. Нарисуйте последовательность деревьев бинарного поиска, которые получаются при поочередном удалении ключей из дерева из упражнения 3.2.1 в порядке их вставки.

3.2.18. Нарисуйте последовательность деревьев бинарного поиска, которые получаются при поочередном удалении ключей из дерева из упражнения 3.2.1 в алфавитном порядке.

3.2.19. Нарисуйте последовательность деревьев бинарного поиска, которые получаются при удалении ключей из дерева из упражнения 3.2.1, если каждый раз удалять ключ из корня.

3.2.20. Докажите, что время выполнения метода `keys()` с двумя аргументами в ДБП из N узлов не более чем пропорциональна высоте дерева плюс количество ключей в диапазоне.

3.2.21. Добавьте в класс `BST` метод `randomKey()`, который возвращает из таблицы имен случайный ключ за время, в худшем случае пропорциональное высоте дерева.

3.2.22. Докажите, что если у узла в ДБП имеются два дочерних узла, то у его преемника нет левого дочернего узла, а у его предшественника — правого.

3.2.23. Является ли операция `delete()` коммутативной? То есть, приводит ли удаление `x` с последующим удалением `y` к тому же результату, что и удаление `y` с последующим удалением `x`?

3.2.24. Докажите, что ни один алгоритм, основанный на сравнениях, не может построить ДБП с помощью менее $\lg(N!) \sim N\lg N$ сравнений.

Творческие задачи

3.2.25. *Идеальная балансировка.* Напишите программу, которая вставляет набор ключей в первоначально пустое ДБП так, что полученное дерево эквивалентно бинарному поиску — т.е. последовательность сравнений, выполняемых при поиске любого ключа в ДБП, совпадает с последовательностью сравнений, выполняемых при бинарном поиске в том же наборе ключей.

3.2.26. *Точные вероятности.* Найдите вероятности появления каждого из деревьев из упражнения 3.2.9 в качестве результата вставки N случайных различных элементов в первоначально пустое дерево.

3.2.27. *Использование памяти.* Сравните требование к памяти в классе `BST` с требованиями к памяти в классах `BinarySearchST` и `SequentialSearchST` для N пар ключ-значение при условиях, описанных в разделе 1.4 (см. упражнение 3.1.21). Учитывайте не память для самих ключей и значений, а ссылки на них. Потом начертите диаграмму для точного использования памяти в ДБП для ключей `String` и значений `Integer` (вроде создаваемых программой `FrequencyCounter`), а затем выведите формулу для объема памяти (в байтах) для ДБП, построенного клиентом `FrequencyCounter` с классом `BST`, и для "Tale of Two Cities".

3.2.28. *Программное кеширование.* Добавьте в класс `BST` сохранение в переменной экземпляров узла, к которому было последнее обращение, чтобы в него можно было попасть за константное время, если операция `put()` или `get()` повторно использует тот же ключ (см. упражнение 3.1.25).

3.2.29. *Проверка бинарного дерева.* Напишите рекурсивный метод `isBinaryTree()`, который принимает в качестве аргумента узел `Node` и возвращает `true`, если счетчик узлов `N` для поддерева с указанным корнем верен, и `false` в противном случае.

3.2.30. *Проверка упорядоченности.* Напишите рекурсивный метод `isOrdered()`, который принимает в качестве аргумента узел `Node` и два ключа `min` и `max` и возвращает `true`, если значения всех ключей в дереве находятся между значениями `min` и `max` (т.е. это наименьший и наибольший ключи в дереве) и свойство упорядоченности ДБП верно для всех поддеревьев указанного дерева; иначе метод должен возвратить `false`.

3.2.31. *Проверка на равные ключи.* Напишите метод `hasNoDuplicates()`, который принимает в качестве аргумента узел `Node` и возвращает `true`, если в бинарном дереве с корнем в указанном узле нет равных ключей, и `false` в противном случае. Считайте, что проверка из предыдущего упражнения уже выполнена.

3.2.32. *Сертификация.* Напишите метод `isBST()`, который принимает в качестве аргумента узел `Node` и возвращает `true`, если указанный узел является корнем дерева бинарного поиска, и `false` в противном случае. *Внимание*: эта задача также труднее, чем может показаться на первый взгляд, поскольку в последних трех упражнениях важен порядок вызова методов.

Решение:

```
private boolean isBST()
{
   if (!isBinaryTree(root)) return false;
   if (!isOrdered(root, min(), max())) return false;
   if (!hasNoDuplicates(root)) return false;
   return true;
}
```

3.2.33. *Проверка выбора и ранга.* Напишите метод, который проверяет, что для всех `i` от `0` до `size()-1` верно, что `i = rank(select(i))`, и для всех ключей в ДБП верно, что `key = select(rank(key))`.

3.2.34. *Связывание.* Реализуйте расширенный API `ThreadedST`, который поддерживает две дополнительные операции, выполняемые за константное время:

- `Key next(Key key)` *ключ, следующий за* `key` *(*`null`*, если* `key` *наибольший)*
- `Key prev(Key key)` *ключ, предшествующий* `key` *(*`null`*, если* `key` *наименьший)*

Для этого добавьте в структуру `Node` поля `pred` и `succ`, содержащие ссылки на предшествующий и последующий узлы, и добавьте в методы `put()`, `deleteMin()`, `deleteMax()` и `delete()` поддержку этих полей.

3.2.35. *Более точный анализ.* Уточните математическую модель, которая бы лучше объясняла экспериментальные результаты из табл. 3.2.2. А именно, покажите, что среднее количество сравнений для успешного поиска в дереве, построенном из случайных ключей, с ростом N приближается к пределу $2\ln N + 2\gamma - 3 \approx \approx 1{,}39 \lg N - 1{,}85$, где $\gamma = 0{,}57721...$ — *константа Эйлера. Совет*: воспользуйтесь анализом быстрой сортировки из раздела 2.3 и тем фактом, что интеграл от $1/x$ асимптотически приближается к $\ln N + \gamma$.

3.2.36. *Итератор.* Возможно ли написать нерекурсивный вариант метода `keys()`, который использует память, пропорциональную высоте дерева (независимо от количества ключей в диапазоне)?

3.2.37. *Поуровневый обход.* Напишите метод `printLevel()`, который принимает в качестве аргумента узел `Node` и выводит ключи из поддерева с корнем в указанном узле по уровням — т.е. в порядке их расстояния до корня, а для узлов с одинаковым расстоянием в порядке слева направо. *Совет*: воспользуйтесь классом `Queue`.

3.2.38. *Вычерчивание дерева.* Добавьте в класс `BST` метод `draw()`, который вычерчивает ДБП в стиле рисунков, приведенных в тексте. *Совет*: используйте переменные экземпляров для хранения координат узлов и рекурсивный метод для задания значений этих переменных.

Эксперименты

3.2.39. *Средний случай.* Экспериментально определите среднее значение и среднеквадратичное отклонение для количества сравнений, выполняемых при попаданиях и промахах в ДБП, построенных в 100 выполнениях вставки N случайных ключей в первоначально пустое дерево, для $N = 10^4$, 10^5 и 10^6. Сравните полученный результат с формулой для среднего значения, приведенной в упражнении 3.2.35.

3.2.40. *Высота.* Экспериментально определите среднюю высоту ДБП, построенных в 100 выполнениях вставки N случайных ключей в первоначально пустое дерево, для $N = 10^4$, 10^5 и 10^6. Сравните полученный результат с формулой $2{,}99 \lg N$, приведенной в тексте.

3.2.41. *Представление массивом.* Разработайте реализацию, в которой ДБП представляется тремя массивами (уже созданными с размером, который задан в конструкторе): один массив для ключей, один — для индексов, соответствующих левым ссылкам, и один — для индексов, соответствующих правым ссылкам. Сравните производительность полученной программы и стандартной реализации.

3.2.42. *Деградация из-за удаления Хиббарда.* Напишите программу, которая принимает из командной строки целое число N, строит случайное ДБП размером N, а затем N^2 раз выполняет удаление случайного ключа (с помощью кода

`delete(select(StdRandom.uniform(N)))` и вставку случайного ключа. После этого измерьте и выведите среднюю длину пути в дереве (длина внутреннего пути, деленная на N, плюс 1). Выполните полученную программу для $N = 10^2$, 10^3 и 10^4 и проверьте (неочевидную) гипотезу, что описанный процесс увеличивает среднюю длину пути в дереве пропорционально квадратному корню из N. Проведите такое же исследование для реализации `delete()`, в которой случайным образом выбирается преемник или предшественник.

3.2.43. *Отношение put/get.* Эмпирически определите отношение времени, которое реализация `BST` тратит на выполнение операций `put()` и операций `get()`, если программа `FrequencyCounter` используется для определения частот появления значений в 1 миллионе случайных целых чисел.

3.2.44. *Графики стоимости.* Доработайте реализацию `BST`, чтобы она выводила графики, подобные графикам в этом разделе, для стоимости каждой операции `put()` во время вычисления (см. также упражнение 3.1.38).

3.2.45. *Замер точного времени.* Добавьте в программу `FrequencyCounter` использование классов `Stopwatch` и `StdDraw` для формирования графиков, где по оси x откладывается количество вызовов `get()` или `put()`, а по оси y — общее время выполнения. Точки графика должны отображать кумулятивное время выполнения после каждого вызова. Выполните полученную программу для "Tale of Two Cities" с реализациями `SequentialSearchST` и `BinarySearchST`, а потом с реализацией `BinarySearchST`, и проанализируйте результаты. *Примечание*: резкие скачки на кривой объясняются *кешированием*, которое выходит за рамки этого вопроса (см. также упражнение 3.1.39).

3.2.46. *Исследование деревьев бинарного поиска.* Найдите значения N, для которых построение таблицы имен на основе ДБП из N случайных ключей `double` становится в 10, 100 и 1000 раз быстрее бинарного поиска. Предскажите значения аналитически и проверьте их экспериментально.

3.2.47. *Среднее время поиска.* Экспериментально определите среднее значение и среднеквадратичное отклонение для средней длины пути к случайному узлу (длина внутреннего пути, деленная на размер дерева, плюс 1) в ДБП, построенном вставкой N случайных ключей в первоначально пустое дерево, для N от 100 до 10000. Для каждого размера дерева выполните 1000 проб. Оформите полученные результаты в виде графика, наподобие приведенного на рис. 3.2.16, и там же вычертите график функции $1{,}39\lg N - 1{,}85$ (см. упражнения 3.2.35 и 3.2.39).

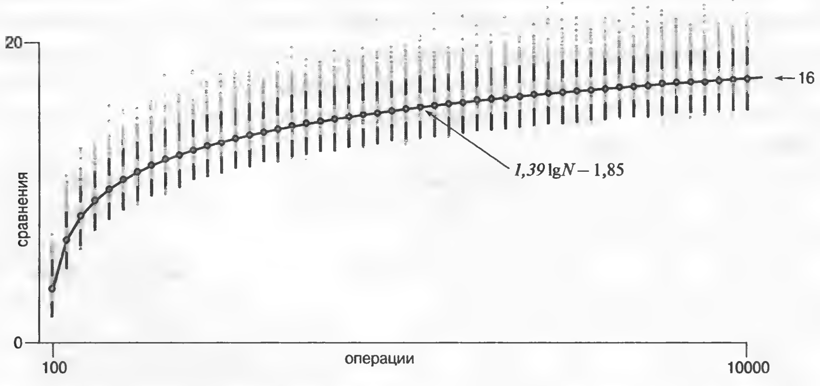


Рис. 3.2.16. Средняя длина пути к случайному узлу в ДБП, построенном из случайных ключей

3.3. Сбалансированные деревья поиска

Алгоритмы, описанные в предыдущем разделе, хорошо работают во многих приложениях, но у них плохая производительность в худшем случае. В данном разделе мы познакомимся с разновидностью деревьев бинарного поиска, в которых стоимость *гарантированно* логарифмическая, независимо от последовательности ключей, из которых это дерево создано. В идеале хотелось бы поддерживать деревья бинарного поиска полностью сбалансированными. В дереве из N узлов желательно, чтобы высота дерева была равна $\sim\lg N$ — тогда все поиски гарантированно будут выполнять $\sim\lg N$ сравнений, как и в случае бинарного поиска (см. утверждение Б). К сожалению, поддержка абсолютного баланса для динамических вставок слишком трудоемка. В этом разделе мы рассмотрим структуру данных, которая несколько смягчает требование обязательной логарифмической производительности не только для операций *вставить* и *найти* из нашего API таблицы имен, но и для всех операций, основанных на упорядоченности (кроме поиска диапазона).

2-3-деревья поиска

Главное, что нужно сделать для гарантии сбалансированности в деревьях поиска — позволить узлам этих деревьев содержать более одного ключа. Наряду с узлами стандартного ДБП (мы будем называть их *2-узлами*, т.к. они содержат две ссылки и один ключ) мы разрешим присутствие *3-узлов*, которые содержат три ссылки и два ключа (рис. 3.3.1). И в 2-узлах, и в 3-узлах каждому интервалу, обусловленному ключами этого узла, соответствует одна ссылка.

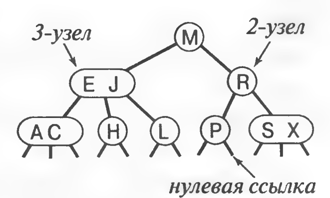


Рис. 3.3.1. *Анатомия 2-3-дерева поиска*

Определение. *2-3-дерево поиска* — это дерево, которое или пусто, или представляет собой:

- *2-узел* с одним ключом (и связанным с ним значением) и двумя ссылками: левая ссылка указывает на 2-3-дерево поиска с меньшими ключами, а правая ссылка указывает на 2-3-дерево с большими ключами;
- *3-узел* с двумя ключами (и связанными с ними значениями) и *тремя ссылками*: левая ссылка указывает на 2-3-дерево поиска с меньшими ключами, средняя ссылка указывает на 2-3-дерево поиска с ключами между ключами узла, а правая ссылка указывает на 2-3-дерево с большими ключами.

Как обычно, ссылка на пустое дерево называется *нулевой ссылкой*.

Идеально сбалансированное 2-3-дерево поиска — такое 2-3-дерево, все нулевые ссылки которого находятся на одном расстоянии от корня. Для краткости термином *2-3-дерево* мы будем называть идеально сбалансированное дерево поиска (в других контекстах это может быть более общая структура). Ниже мы ознакомимся с эффективными способами определения и реализации базовых операций с 2-узлами, 3-узлами и 2-3-деревьями; а пока просто предположим, что все это возможно, и посмотрим, как можно использовать новые конструкции в деревьях поиска.

Поиск

Алгоритм поиска ключей в 2-3-дереве непосредственно обобщает алгоритм поиска в ДБП (рис. 3.3.2). Чтобы узнать, присутствует ли ключ в дереве, мы сравниваем его с ключами в корне. Если он равен одному из них, то это попадание. Иначе мы спускаемся из корня в поддерево, соответствующее интервалу значений ключей, который может содержать искомый ключ. Если ссылка нулевая, то это промах, иначе выполняется рекурсивный поиск в поддереве.

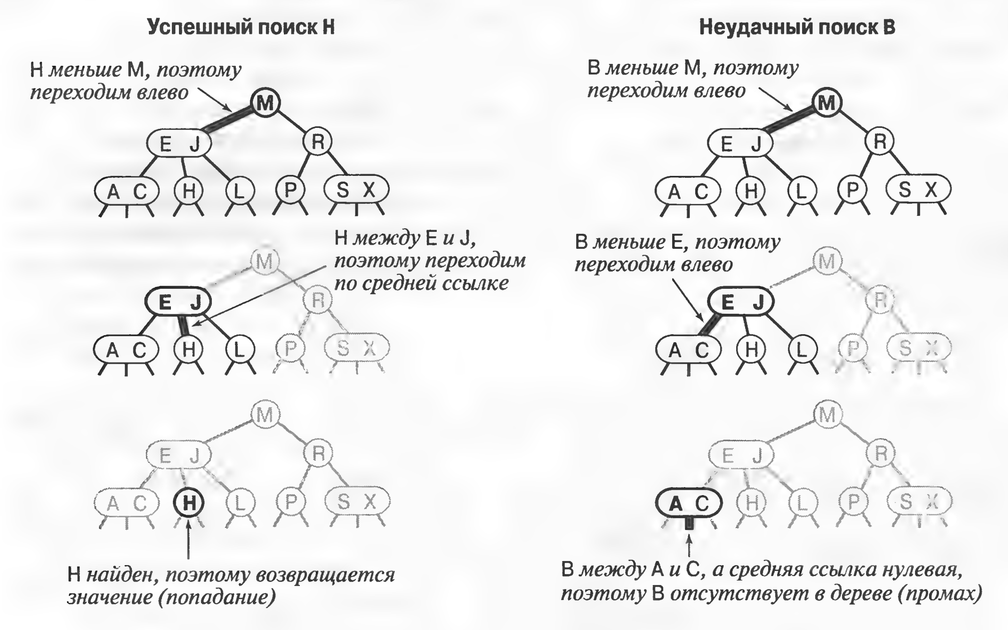


Рис. 3.3.2. *Попадание (слева) и промах (справа) в 2-3-дереве*

Вставка в 2-узел

Чтобы выполнить вставку нового узла в 2-3-дерево, можно провести неудачный поиск, а затем привязать узел внизу, как в ДБП, но тогда полученное дерево может оказаться не идеально сбалансированным. Основное преимущество 2-3-деревьев в том, что в нем можно выполнять вставки и все-таки сохранять идеальный баланс. Если узел, на котором остановился поиск, является 2-узлом, то решить такую задачу нетрудно: нужно просто заменить этот узел 3-узлом, содержащим старый ключ и вставленный ключ (рис. 3.3.3). Но если узел, на котором остановился поиск, является 3-узлом, то придется немного повозиться.

Вставка в дерево, состоящее из одного 3-узла

Прежде чем приступить к общему случаю, для начала предположим, что нужно вставить ключ в маленькое 2-3-дерево, состоящее из единственного 3-узла. Такой узел уже содержит два ключа, и поэтому в нем нет места для еще одного. Для выполнения вставки мы временно поместим ключ в *4-узел* — естественное расширение нашего типа узла, в котором три ключа и четыре ссылки. Такой узел удобен тем, что его легко преобразовать в 2-3-дерево, состоящее из трех 2-узлов: один со средним ключом (в корне), один с меньшим из трех ключей (на него указывает левая ссылка корня), и один с большим из трех ключей (на него указывает правая ссылка корня), см. рис. 3.3.4.

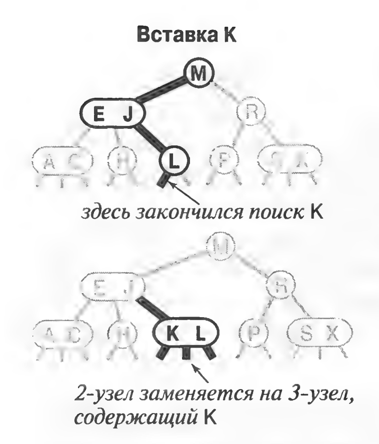


Рис. 3.3.3. Вставка в 2-узел

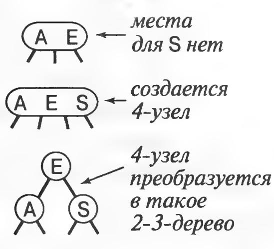


Рис. 3.3.4. Вставка в единственный 3-узел

Полученное дерево представляет собой ДБП и одновременно идеально сбалансированное 2-3-дерево поиска, в котором все нулевые ссылки находятся на одинаковом расстоянии от корня. До вставки высота дерева была равна 0, а после вставки стала равной 1. Данный случай довольно простой, но он демонстрирует рост высоты 2-3-деревьев.

Вставка в 3-узел, родитель которого является 2-узлом

В качестве еще одного подготовительного шага предположим, что поиск завершился в 3-узле внизу дерева, родителем которого является 2-узел. В этом случае также можно найти место для нового ключа, *не нарушая идеального баланса в дереве*. Для этого, как и раньше, создается 4-узел и разбивается на части, но вместо создания нового узла для среднего ключа этот средний ключ перемещается в родительский узел в качестве среднего ключа. Это преобразование выглядит как замена ссылки на старый 3-узел в родительском узле средним ключом, а левая и правая ссылки указывают на новые 2-узлы (рис. 3.3.5). По нашему предположению, место в родительском узле для этого есть: родительский узел был 2-узлом (с одним ключом и двумя ссылками) и стал 3-узлом (с двумя ключами и тремя ссылками). Кроме того, описанное преобразование не меняет определяющих свойств (идеально сбалансированных) 2-3-деревьев. Дерево остается упорядоченным, т.к. средний ключ перемещен в родительский узел, и оно идеально сбалансировано: все его нулевые ссылки были на одном расстоянии от корня до вставки и остались на одном расстоянии от корня после вставки. Обязательно разберитесь в этом преобразовании, ведь на нем держится вся динамика 2-3-дерева.



Рис. 3.3.5. Вставка в 3-узел, родитель которого является 2-узлом

Вставка в 3-узел, родитель которого является 3-узлом

Теперь предположим, что поиск завершился в узле, родителем которого является 3-узел. Здесь так же создается временный 4-узел, который разбивается с передачей среднего ключа родительскому узлу. Поскольку родитель также является 3-узлом, он тоже заменяется временным 4-узлом, содержащим средний ключ от предыдущего разбиения 4-узла. И *с этим узлом выполняется точно такое же преобразование.*

То есть новый 4-узел разбивается со вставкой среднего ключа *его* родителю. Дальше понятно: этот процесс — разбиение 4-узлов с передачей средних ключей на уровень выше — продолжается вверх по дереву, пока не встретится 2-узел (и этот 2-узел заменяется 3-узлом, который уже не нужно разбивать) или 3-узел в корне дерева (рис. 3.3.6).

Разбиение корня

Если на всем пути от места вставки до корня находились только 3-узлы, то, в конце концов, образуется временный 4-узел в корне. В этом случае можно поступить так же, как и при вставке в дерево, состоящее из единственного 3-узла. Временный 4-узел разбивается на три 2-узла, и высота дерева увеличивается на 1 (рис. 3.3.7). Это последнее преобразование сохраняет идеальный баланс, т.к. оно выполняется в корне.

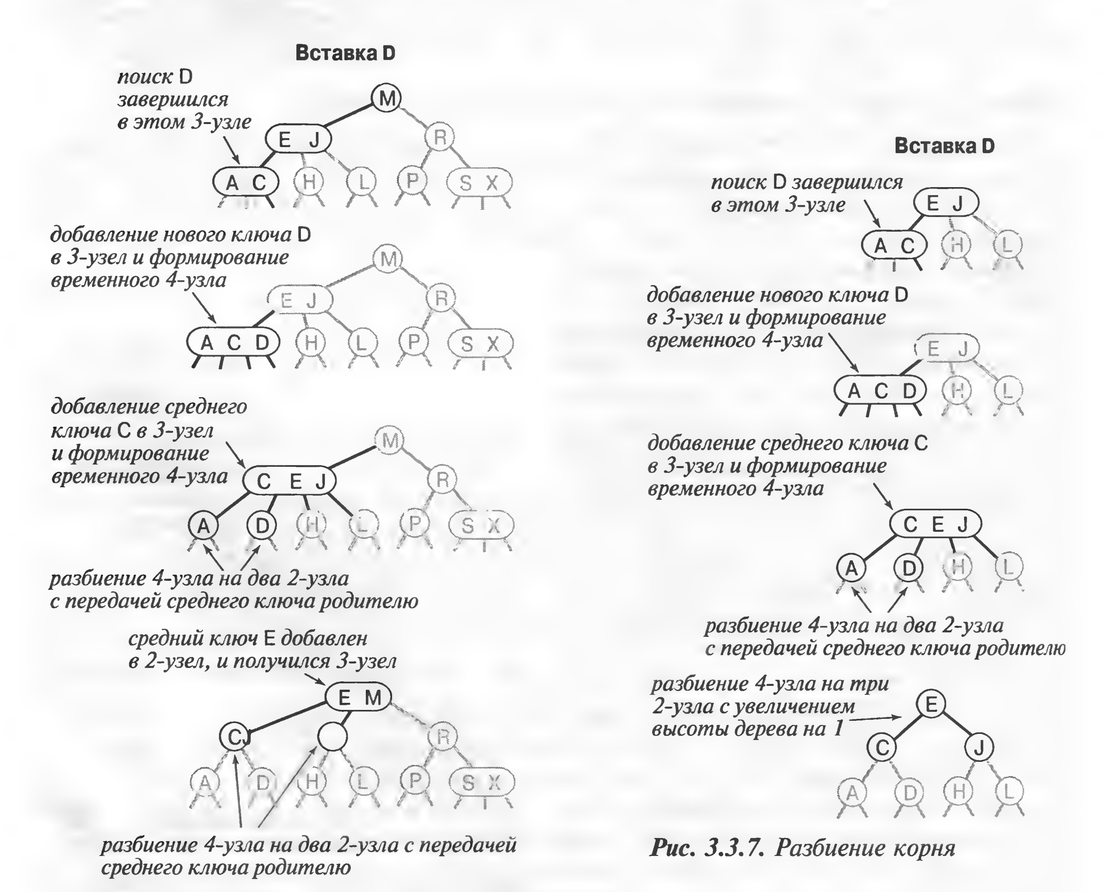


Рис. 3.3.6. Вставка в 3-узел, родитель которого является 3-узлом

Рис. 3.3.7. Разбиение корня

Локальные преобразования

Разбиение временного 4-узла в 2-3-дереве требует выполнения одного из шести преобразований, которые приведены на рис. 3.3.8. 4-корень может быть корнем, левым/правым дочерним узлом 2-узла или левым/правым дочерним узлом 3-узла. Смысл алгоритма вставки в 2-3-дерево в том, что все эти преобразования чисто *локальны*: требуется просматривать или изменять только очень небольшую часть дерева в районе заданных узлов и ссылок.

Количество ссылок, изменяемых в каждом преобразовании, ограничено небольшой константой. В частности, эти преобразования эффективны для указанных подструктур *в любом месте дерева*, а не только внизу. Каждое из таких преобразований передает из 4-узла в родительский узел один из его ключей, а затем перестраивает ссылки, не затрагивая другие части дерева.

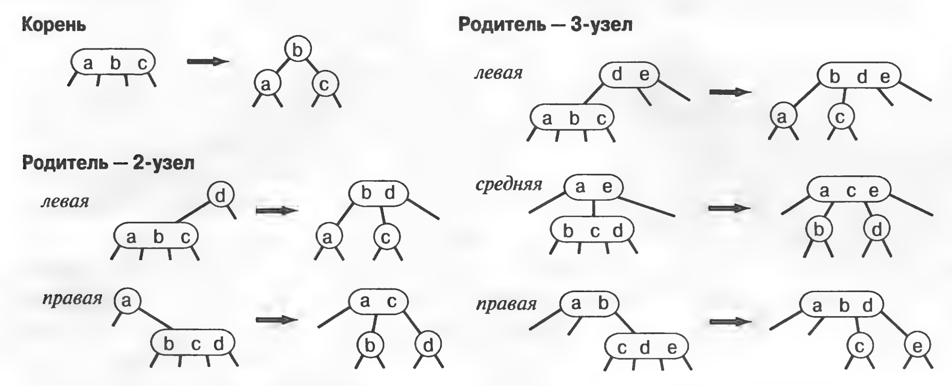


***Рис. 3.3.8.** Разбиение временного 4-узла в 2-3-дереве (сводка)*

Глобальные свойства

Кроме того, эти *локальные* преобразования сохраняют *глобальные* свойства упорядоченности и идеального баланса: количество ссылок на пути от корня до любой нулевой ссылки одно и то же. На рис. 3.3.9 приведена полная диаграмма, которая демонстрирует, что 4-узел является средним дочерним узлом 3-узла. Если до преобразования длина каждого пути от корня до нулевой ссылки была равна h, то она остается равной h и после преобразования. *Каждое преобразование сохраняет это свойство* — даже при разбиении 4-узла на два 2-узла и замене родительского 2-узла на 3-узел или 3-узла на 4-узел. При разбиении корня на три 2-узла длина каждого пути от корня до нулевой ссылки увеличивается на 1. Если вы еще не убеждены в этом, проработайте упражнение 3.3.7, где нужно добавить к рис. 3.3.9 пять других вариантов, иллюстрирующих тот же принцип. Понимание, что каждое локальное преобразование сохраняет упорядоченность и идеальный баланс во всем дереве, является ключом к пониманию алгоритма в целом.

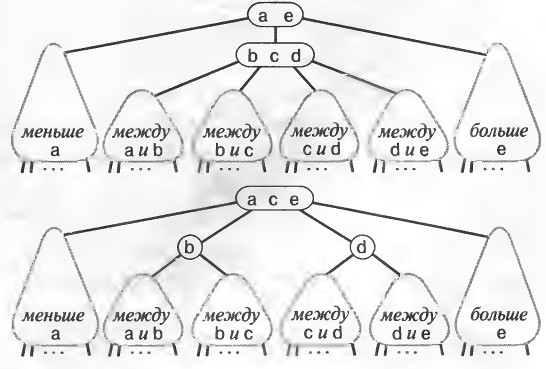


***Рис. 3.3.9.** Разбиение 4-узла — это локальное преобразование, которое сохраняет упорядоченность и идеальный баланс*

В отличие от стандартных ДБП, которые разрастаются сверху вниз, 2-3-деревья разрастаются снизу. Не пожалейте времени и внимательно изучите рис. 3.3.10: на нем изображена последовательность 2-3-деревьев, сгенерированная нашим стандартным клиентом индексации, и 2-3-деревьев, сгенерированная при вставке тех же ключей в порядке возрастания. После этого вы будете хорошо понимать принцип построения 2-3-деревьев. Вспомните, что при вставке 10 ключей в ДБП в порядке возрастания формируется дерево худшего случая высотой 9. В случае 2-3-деревьев высота равна 2.

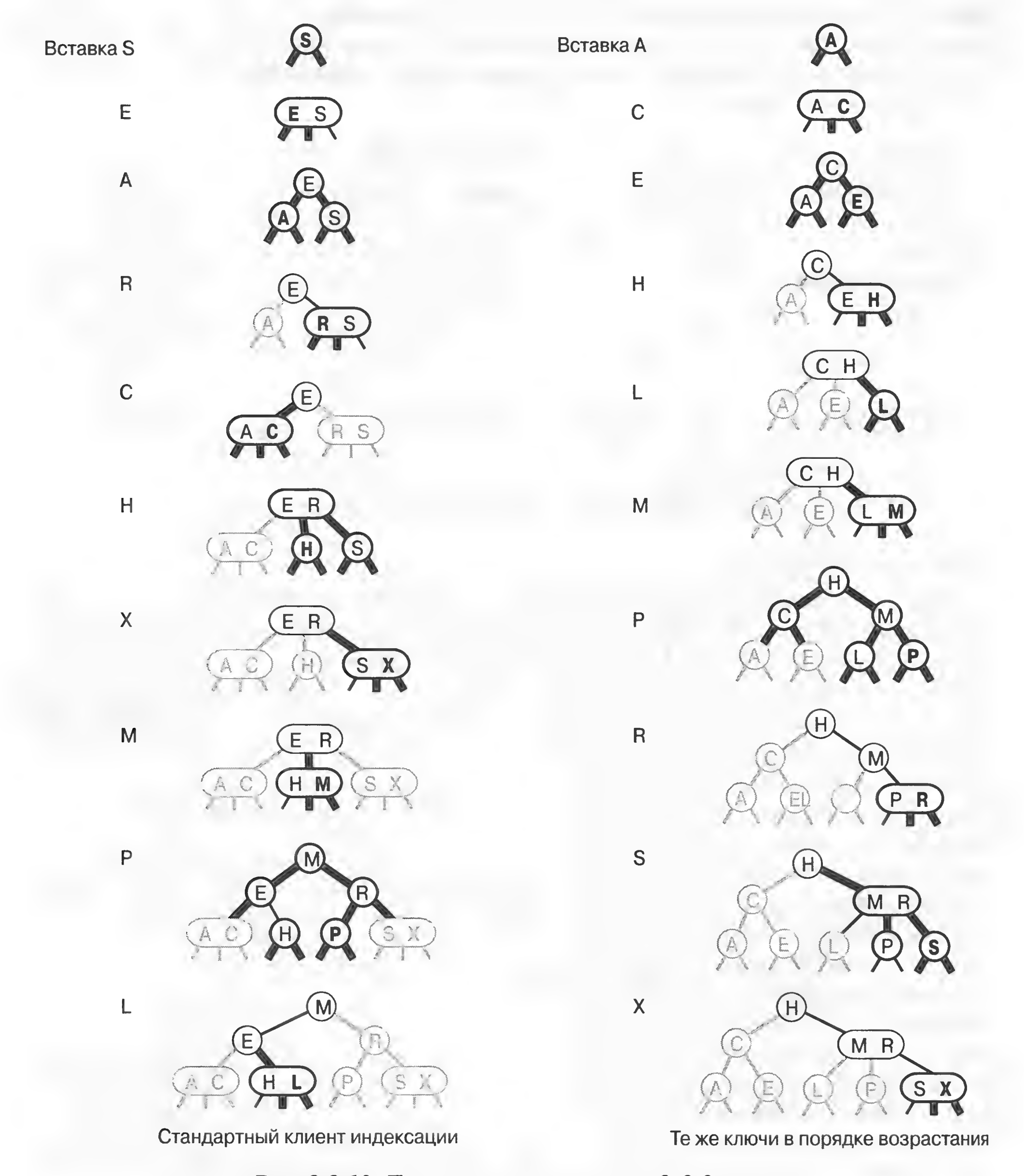


Рис. 3.3.10. Трассировки построения 2-3-деревьев

Приведенного выше описания достаточно, чтобы определить реализацию таблицы имен, основанную на 2-3-деревьях. Анализ 2-3-деревьев отличается от анализа ДБП, потому что здесь нас интересует производительность *в худшем случае*, а не в среднем (когда производительность прогнозируется на основе модели случайных ключей). В реализациях таблиц имен обычно нет возможности управления порядком, в котором клиенты вставляют ключи в таблицу, а анализ худшего случая — один из способов предоставить гарантии производительности.

Утверждение Е. Операции поиска и вставки в 2-3-дереве с N ключами гарантированно просматривают не более $\lg N$ узлов.

Доказательство. Высота 2-3-дерева из N узлов находится между $\lfloor \log_3 N \rfloor = \lfloor (\lg N)/(\lg 3) \rfloor$ (если дерево содержит только 3-узлы) и $\lfloor \lg N \rfloor$ (если дерево содержит только 2-узлы) (см. упражнение 3.3.4).

Итак, мы можем гарантировать хорошую производительность при работе с 2-3-деревьями даже в худшем случае. Время, необходимое для выполнения любой операции в каждом узле, ограничено константой, и обе операции просматривают узлы только на одном пути, поэтому общая стоимость любого поиска или вставки гарантированно логарифмична. Сравнение 2-3-дерева на рис. 3.3.11 и ДБП, построенного из тех же ключей (см. рис. 3.2.8), показывает, что идеально сбалансированное 2-3-дерево остается на удивление плоским. Например, высота 2-3-дерева, содержащего 1 миллиард ключей, находится в пределах от 19 до 30. Замечательно, что произвольные операции поиска и вставки среди 1 миллиарда ключей можно гарантированно выполнять с помощью просмотра не более 30 узлов.

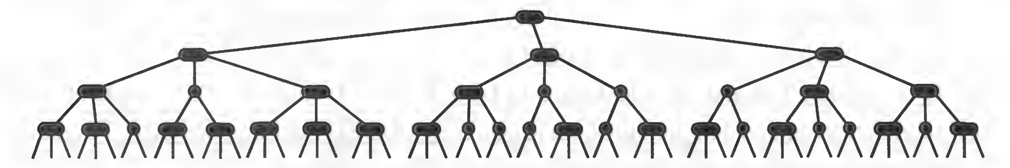


Рис. 3.3.11. Типичное 2-3-дерево, построенное из случайных ключей

Однако мы прошли лишь полпути до реализации. В принципе можно написать код, выполняющий преобразования в различных типах данных, представляющих 2- и 3-узлы, однако большинство описанных нами задач неудобно решать напролом, т.к. придется рассматривать много различных случаев. Понадобятся два различных вида узлов, сравнение искомых ключей с каждым ключом узла, копирование ссылок и другой информации из одного вида узлов в другой, преобразование узлов из одного вида в другой и т.д. Для этого потребуется не только значительный объем кода, но и время, и в результате такой алгоритм будет работать медленнее поиска и вставки в стандартных ДБП. Конечно, основное назначение балансировки — гарантия от худшего случая, но хотелось бы добиться этого не слишком большой ценой. К счастью, как мы увидим, есть возможность выполнять все преобразования единообразно и с небольшими дополнительными затратами.

Красно-черные ДБП

Только что описанный алгоритм вставки для 2-3-деревьев нетрудно понять; а сейчас мы покажем, что его нетрудно и реализовать. Мы рассмотрим простое представление — *красно-черные ДБП*, для которых существует естественная реализация. Кода для этого

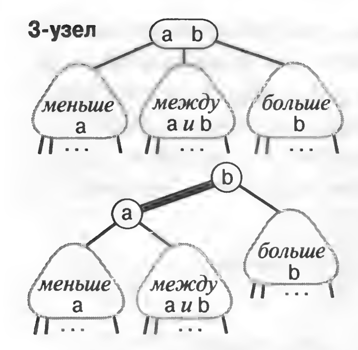


Рис. 3.3.12. Представление 3-узла двумя 2-узлами с красной связью, направленной влево

потребуется немного, но вот чтобы понять, как и почему этот код выполняет все нужные действия, потребуется хорошо разобраться в процессе.

Кодирование 3-узлов

Чтобы реализовать красно-черные ДБП, предназначенные для кодирования 2-3-деревьев, мы начнем с обычных ДБП (которые состоят исключительно из 2-узлов) и добавим в них дополнительную информацию для кодирования 3-узлов. Мы будем различать ссылки двух видов: *красные* связи, которые соединяют два 2-узла, образуя 3-узел (на рисунках это толстые линии), и *черные* связи, которые формируют само дерево. Конкретнее, мы будем представлять 3-узлы двумя 2-узлами, соединенными красной связью, которая *направлена влево* (один из 2-узлов является левым дочерним узлом другого, рис. 3.3.12). Одно из преимуществ такого представления в том, что наш код метода `get()` для поиска в стандартном ДБП можно использовать *без изменений*. Для любого заданного 2-3-дерева можно непосредственно построить соответствующее ДБП, просто преобразовав узлы так, как только что описано. Деревья бинарного поиска, которые таким образом представляют 2-3-деревья, мы будем называть *красно-черными ДБП*.

Эквивалентное определение

Можно поступить по-другому — *определить* черно-красные деревья как деревья бинарного поиска, которые содержат красные и черные ссылки и удовлетворяют следующим трем ограничениям.

- Красные ссылки направлены влево.
- Ни у одного узла нет двух красных ссылок.
- Дерево *идеально сбалансировано в черном цвете*: каждый путь от корня к нулевой ссылке содержит одно и то же количество черных ссылок.

Между так определенными красно-черными ДБП и 2-3-деревьями существует взаимно однозначное соответствие (рис. 3.3.13).

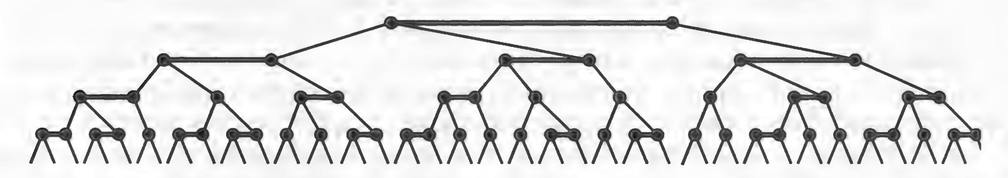


Рис. 3.3.13. Красно-черное дерево с горизонтальными красными ссылками, представляющее 2-3-дерево

Взаимно однозначное соответствие

Если в красно-черном ДБП начертить все красные связи горизонтально, то все нулевые ссылки будут находиться на одинаковом расстоянии от корня, а если сжать узлы, соединенные красными связями, получится 2-3-дерево. И наоборот: если нарисовать 3-узлы в 2-3-дереве в виде 2-узлов, соединенных красной ссылкой, направленной влево, то ни у одного узла не будет двух красных связей, и дерево будет иметь идеальный черный

баланс, т.к. черные ссылки соответствуют ссылкам в 2-3-дереве, которое идеально сбалансировано по определению. Вне зависимости от определения, красно-черные ДБП являются *и* деревьями бинарного поиска, *и* 2-3-деревьями (рис. 3.3.14). Значит, если мы сможем реализовать алгоритм вставки в 2-3-дерево, который поддерживает взаимно однозначное соответствие, то мы возьмем лучшее от обоих видов деревьев: простой и эффективный метод поиска от обычных ДБП и эффективный метод вставки с балансировкой от 2-3-деревьев.

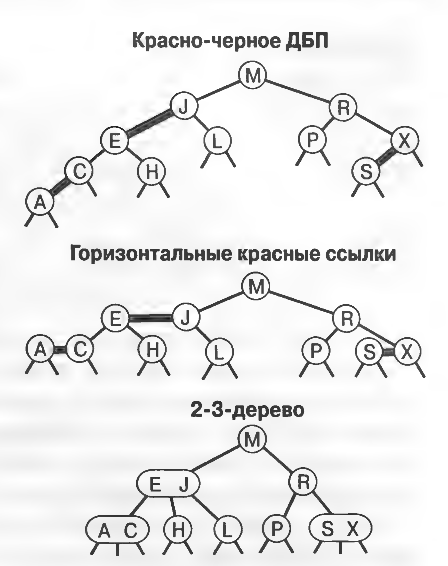


Рис. 3.3.14. *Взаимно однозначное соответствие между красно-черными ДБП и 2-3-деревьями*

Представление цвета

Поскольку на каждый узел указывает точно одна ссылка (от родительского узла), то удобно кодировать цвет связи в *указываемых узлах*. Для этого в тип данных `Node` (см. рис. 3.3.15) можно добавить логическую переменную экземпляров `color`, которая равна `true`, если связь от родительского узла красная, и `false` — если черная. Нулевые ссылки будем считать черными. Для большей понятности нашего кода мы определим константы `RED` и `BLACK`, которые будут полезны для установки и проверки значений этой переменной. А приватный метод `isRed()` позволит нам проверять цвет ссылки на узел из его родителя. Когда мы будем говорить о цвете узла, это будет означать цвет указывающей на него ссылки, и наоборот.

```
private static final boolean RED   = true;
private static final boolean BLACK = false;
private class Node
{
   Key key;               // ключ
   Value val;             // связанные с ключом данные
   Node left, right;      // поддеревья
   int N;                 // количество узлов в данном поддереве
   boolean color;         // цвет ссылки на данный узел

   Node(Key key, Value val, int N, boolean color)
   {
      this.key   = key;
      this.val   = val;
      this.N     = N;
      this.color = color;
   }
}

private boolean isRed(Node x)
{
   if (x == null) return false;
   return x.color == RED;
}
```

h.left.color равно RED
h
h.right.color равно BLACK

Рис. 3.3.15. *Представление узла в красно-черных ДБП*

Развороты

В реализации, которую мы рассмотрим, во время выполнения операций могут появляться красные связи, направленные вправо, или две красные связи подряд. Но эти ситуации будут исправляться еще до завершения операции с помощью специальной операции *разворота*, которая изменяет ориентацию красных ссылок.

Рассмотрим вначале правую красную связь, которую необходимо развернуть, чтобы она стала направленной влево (рис. 3.3.16). Эта операция называется *разворотом влево*. Мы оформим ее в виде метода, который принимает в качестве аргумента ссылку на красно-черное ДБП и считает, что эта ссылка указывает на узел Node с правой красной связью. Метод выполняет необходимые перестановки и возвращает ссылку на корень красно-черного ДБП с тем же набором ключей, но с *левой* красной связью. Если сравнить каждую строку кода с рисунками до и после на диаграмме, вы увидите, что эта операция совсем не сложна для понимания: в ней выполняется переключение с корня, содержащего меньший из двух ключей, на корень, содержащий больший из двух ключей. Реализация *разворота вправо*, который преобразует красную связь, направленную влево, в красную ссылку, направленную вправо, выполняется с помощью такого же кода, в котором направления вправо и влево поменялись местами (рис. 3.3.17).

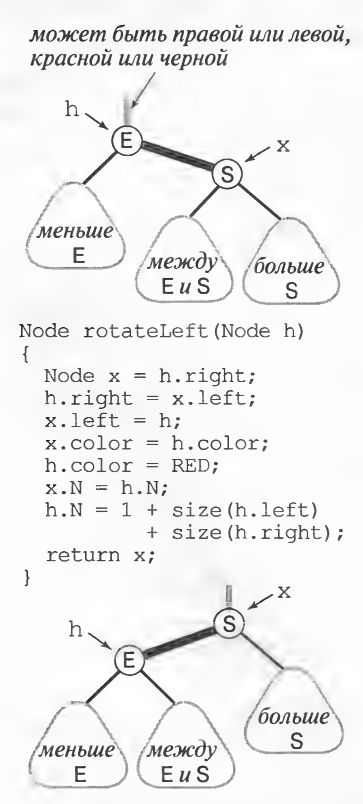


Рис. 3.3.16. Разворот влево (правая ссылка узла h)

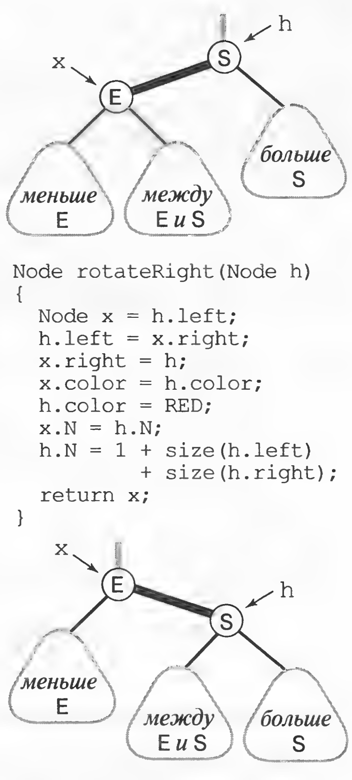


Рис. 3.3.17. Разворот вправо (левая ссылка узла h)

Переустановка ссылки в родительском узле после разворота

Результатом любого разворота — влево или вправо — является ссылка. Ссылку, возвращенную вызовом `rotateRight()` или `rotateLeft()`, мы используем для установки соответствующей ссылки в родительском узле (или в корне дерева). Это может быть правая или левая ссылка, но по ней можно переустановить ссылку в родительском узле. Такая ссылка может быть как красной, так и черной: методы `rotateRight()` и `rotateLeft()` сохраняют ее цвет, присваивая значение `x.color` полю `h.color`. При этом в дереве могут появиться две красные связи подряд, но наши алгоритмы с помощью разворотов исправляют такие ситуации сразу после их возникновения. Например, код

```
h = rotateLeft(h);
```

разворачивает влево красную ссылку, которая указывает вправо от узла `h`, в результате чего `h` будет указывать на корень полученного поддерева. Новое поддерево будет содержать все те же узлы, что и поддерево, на которое указывала ссылка `h` до разворота, но с другим корнем. Легкость написания подобного кода и является основной причиной, по которой мы используем рекурсивные реализации методов ДБП: как мы увидим, выполнение разворотов несложно добавить к обычной вставке.

Развороты позволяют поддерживать взаимно однозначное соответствие между 2-3-деревьями и красно-черными деревьями при вставке новых ключей, т.к. они сохраняют два определяющих свойства красно-черных ДБП: *упорядоченность* и *идеальный черный баланс*. Это означает, что мы можем использовать развороты в красно-черных деревьях, не заботясь о том, что нарушится упорядоченность или идеальный черный баланс. Сейчас мы рассмотрим, как развороты помогают сохранять два других определяющих свойства красно-черных ДБП — отсутствие двух красных связей подряд на любом пути и отсутствие красных связей, направленных вправо. Начнем с нескольких простых случаев.

Вставка в единственный 2-узел

Красно-черное ДБП с одним ключом — это просто 2-узел. Для вставки второго ключа сразу же требуется операция разворота. Если новый ключ меньше ключа в дереве, то просто создается новый (красный) узел с новым ключом, и все: получено красно-черное ДБП, эквивалентное единственному 3-узлу. Но если новый ключ больше ключа в дереве, то добавление нового (красного) узла приводит к появлению правой красной ссылки, и для завершения вставки нужно выполнить оператор `root = rotateLeft(root);`, который разворачивает красную ссылку влево и изменяет ссылку на корень дерева. В обоих случаях в результате получается красно-черное представление единственного 3-узла с двумя ключами, одной левой красной ссылкой и черной высотой, равной 1 (рис. 3.3.18).

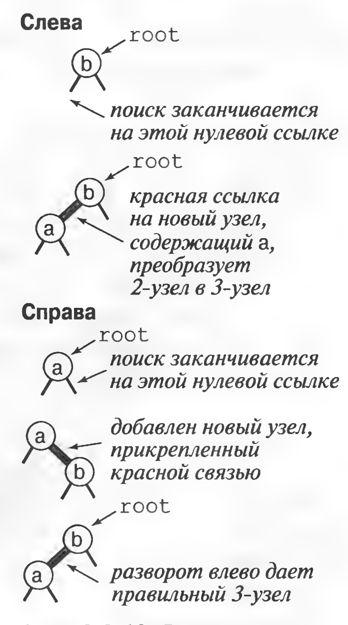


Рис. 3.3.18. Вставка в единственный 2-узел (два варианта)

Вставка в 2-узел на нижнем уровне

В красно-черное ДБП ключи вставляются так же, как и в обычное ДБП. Новый узел добавляется на нижнем уровне (с учетом упорядоченности), но он всегда соединяется со своим родителем красной связью. Если родитель является 2-узлом, то выполняется одна из двух описан-

ных выше процедур. Если новый узел прикреплен к левой ссылке, его родитель просто становится 3-узлом, а если к правой — получится 3-узел, направленный не в ту сторону, и нужен еще завершающий разворот (рис. 3.3.19).





Рис. 3.3.19. Вставка в 2-узел на нижнем уровне

Вставка в дерево с двумя ключами (в 3-узел)

Этот случай возможен в трех вариантах: новый ключ может быть меньше обоих ключей в дереве, между ними или больше обоих ключей. В любом варианте получается узел с двумя красными связями, и нужно исправить это положение.

- Самый простой вариант — когда новый ключ *больше* двух ключей в дереве; поэтому он присоединяется к самой правой связи 3-узла, и получается сбалансированное дерево со средним ключом в корне, к красным связям которого прикреплены меньший и больший ключи. Если изменить цвета этих связей с красного на черный, то получится сбалансированное дерево высотой 2 с тремя узлами — а это как раз то, что нужно для поддержания взаимно однозначного соответствия с 2-3-деревьями. Два других варианта, в конце концов, приводятся к этому же варианту.
- Если новый ключ *меньше* двух ключей в дереве и проходит по левой связи, то получаются две красные связи подряд, и обе левые. Этот случай можно привести к предыдущему варианту (средний ключ в корне, присоединенный к двум другим красными связями), развернув вправо верхнюю связь.
- Если новый ключ попадает *между* двумя ключами в дереве, у нас опять две красные связи подряд (правая под левой). Этот случай приводится к предыдущему (две левые красные связи подряд) разворотом влево нижней связи.

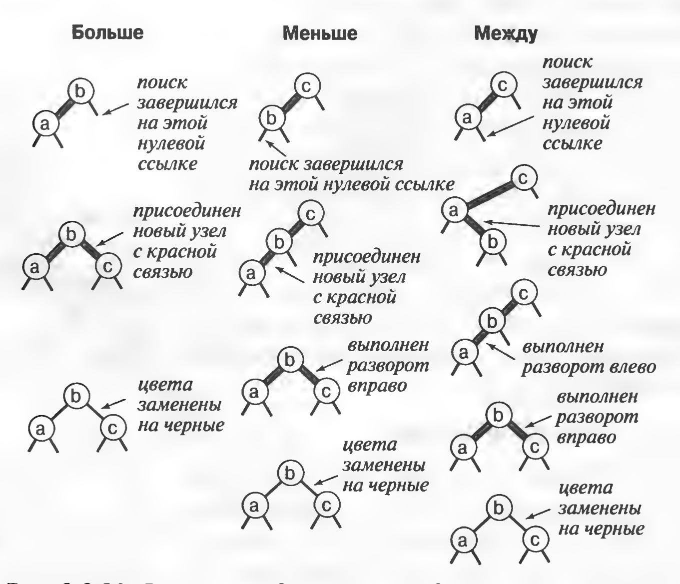


Рис. 3.3.20. Вставка в единственный 3-узел (три варианта)

В любом случае для получения нужного результата необходимо выполнить ноль, один или два разворота, а затем изменить цвета двух дочерних узлов корня (рис. 3.3.20). Как и в случае 2-3-деревьев, *разберитесь, как работают эти преобразования*, т.к. они являются ключом к динамике красно-черных деревьев.

Изменение цветов

Для изменения цветов двух красных дочерних узлов используется метод `flipColors()`, показанный на рис. 3.3.21. Кроме изменения цветов дочерних узлов с красного на черный, меняется еще и цвет родительского узла — с черного на красный. Важно, что данная операция, как и развороты, является локальным преобразованием, которое *сохраняет идеальный черный баланс* в дереве. А, кроме того, это непосредственно приводит нас к полной реализации, которая будет описана чуть ниже.

```
void flipColors(Node h)
{
   h.color = RED;
   h.left.color = BLACK;
   h.right.color = BLACK;
}
```

Рис. 3.3.21. *Изменение цветов для разбиения 4-узла*

Сохранение черной окраски корня

В только что рассмотренном случае (вставка в единственный 3-узел) изменение цвета приводит к окраске корня в красный цвет. Это может произойти и в более крупных деревьях. Строго говоря, красный корень означает, что он является частью 3-узла, но это не так, поэтому после каждой вставки нужно перекрашивать корень в черный цвет. Учтите, что при изменении цвета корня с красного на черный высота дерева увеличивается на 1.

Вставка в 3-узел на нижнем уровне

Теперь предположим, что новый узел добавляется на нижнем уровне в 3-узел. Здесь возникают те же три варианта, которые уже были рассмотрены выше (рис. 3.3.22). Новая связь прикрепляется либо к правой связи 3-узла (в этом случае достаточно просто изменить цвета), либо к левой связи 3-узла (тогда нужно развернуть верхнюю связь вправо и изменить цвета), либо к средней связи 3-узла (тут понадобится развернуть влево нижнюю связь, потом развернуть вправо верхнюю связь, а затем изменить цвета). Из-за изменения цветов ссылка на средний узел становится красной, она передается вверх родительскому узлу — и мы оказываемся в той же ситуации, но по отношению к родителю, что можно исправить подъемом по дереву.

Подъем красной связи по дереву

Алгоритм вставки в 2-3-дерево выполняет разбиение 3-узла и поднимает средний ключ для вставки в родительский узел — и так, пока не встретится 2-узел или корень. В любом из рассмотренных случаев мы выполняем одну и ту же последовательность действий: после выполнения необходимых разворотов мы меняем цвета, после чего средний узел становится красным. С точки зрения родительского узла эту ставшую красной связь можно обработать точно так же, как и красную связь из-за присоединения нового узла: красная связь передается среднему узлу.

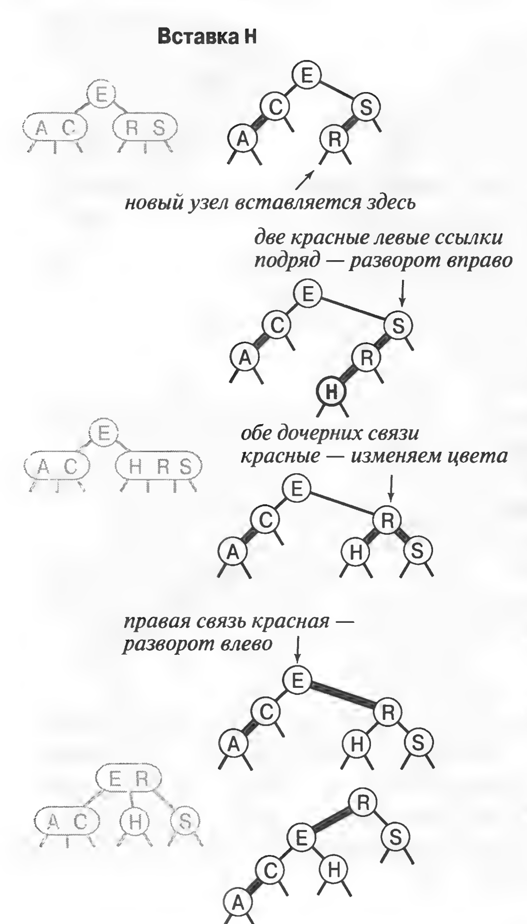


Рис. 3.3.22. Вставка в 3-узел на нижнем уровне

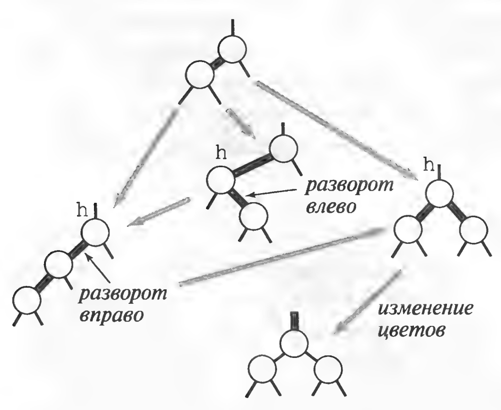


Рис. 3.3.23. Подъем красной ссылки в красно-черном ДБП

Три варианта, представленные на рис. 3.3.23, в точности отображают операции, необходимые в красно-черном дереве для реализации действий с ключами при вставке в 2-3-дереве: для вставки в 3-узел создается временный 4-узел, потом он разбивается, и средняя связь передается в средний ключ родительского узла. Этот процесс подъема красной связи по дереву продолжается до встречи 2-узла или корня.

Итак, взаимно однозначное соответствие между 2-3-деревьями и красно-черными деревьями бинарного поиска можно поддерживать при вставке ключей, выполняя три простые операции — разворот влево, разворот вправо и изменение цветов. Вставку можно выполнить с помощью следующих операций, одна за другой, в каждом узле на пути подъема по дереву от точки вставки.

- Если правый дочерний узел красный, а левый — черный, выполнить разворот влево.
- Если и левый дочерний узел, и его дочерний узел красные, выполнить разворот вправо.
- Если и левый, и правый дочерние узлы красные, изменить цвета.

Не пожалейте времени на проверку, что эта последовательность справляется со всеми описанными выше случаями. Обратите внимание, что первая операция выполняет и разворот, необходимый для наклона 3-узла влево, если его родитель является 2-узлом, и разворот, необходимый для направления нижней связи влево, если новая красная связь является средней связью в 3-узле.

Реализация

Операции балансировки должны выполняться при продвижении *вверх* по дереву от точки вставки, поэтому их нетрудно выполнять в нашей стандартной рекурсивной реализации: они просто выполняются после рекурсивных вызовов, как показано в алгоритме 3.4 (листинг 3.3.1). Каждую из трех операций, перечисленных в предыдущем

разделе, можно выполнить с помощью единственного оператора if, который проверяет цвета двух узлов в дереве. Несмотря на небольшой объем кода, эту реализацию довольно сложно понять без двух вспомогательных уровней абстракции — 2-3-деревьев и красно-черных ДБП. Ценой проверки цветов от трех до пяти узлов (и, возможно, выполнения одного-двух разворотов и изменения цветов, если результаты проверок положительны) мы получаем деревья бинарного поиска с почти идеальным балансом.

Листинг 3.3.1. Алгоритм 3.4. Вставка в красно-черных ДБП

```
public class    RedBlackBST<Key extends Comparable<Key>, Value>
{
  private Node root;

  private class Node // Узел ДБП с битом цвета (см. рис. 3.3.15)

  private boolean isRed(Node h)        // См. рис. 3.3.15.
  private Node rotateLeft(Node h)      // См. рис. 3.3.16.
  private Node rotateRight(Node h)     // См. рис. 3.3.17.
  private void flipColors(Node h)      // См. рис. 3.3.21.

  private int size()                   // См. листинг 3.2.1.

  public void put(Key key, Value val)
  { // Поиск ключа. Если найден, изменяется значение;
    // если нет — увеличивается таблица.
    root = put(root, key, val);
    root.color = BLACK;
  }
  private Node put(Node h, Key key, Value val)
  {
    if (h == null)  // Выполняется обычная вставка с красной связью с родителем.
      return new Node(key, val, 1, RED);

    int cmp = key.compareTo(h.key);
    if      (cmp < 0) h.left  = put(h.left,  key, val);
    else if (cmp > 0) h.right = put(h.right, key, val);
    else h.val = val;

    if (isRed(h.right) && !isRed(h.left))    h = rotateLeft(h);
    if (isRed(h.left) && isRed(h.left.left)) h = rotateRight(h);
    if (isRed(h.left) && isRed(h.right))     flipColors(h);

    h.N = size(h.left) + size(h.right) + 1;
    return h;
  }
}
```

Код рекурсивного метода put() для красно-черных ДБП очень похож на метод put() для обычных ДБП, но в нем после рекурсивных вызовов имеются три оператора if, которые поддерживают в дереве почти идеальный баланс с помощью взаимно однозначного соответствия с 2-3-деревьями при подъеме по пути поиска. Первый оператор if разворачивает влево любой 3-узел, направленный вправо (или направленную вправо красную связь под временным 4-узлом); второй разворачивает вправо верхнюю связь во временный 4-узел с двумя направленными влево красными связями; а третий изменяет цвета, поднимая красную связь вверх по дереву (см. текст).

Трассировки для нашего стандартного клиента индексации и для тех же ключей, вставляемых в порядке возрастания, приведены на рис. 3.3.24. Анализ этих примеров исключительно на основе наших трех операций в красно-черных деревьях — очень полезное упражнение. Еще одно полезное упражнение — проверка соответствия с 2-3-деревьями, которое поддерживает алгоритм (используя рис. 3.3.10 для тех же ключей). В обоих случаях вы можете проверить свое понимание алгоритма, рассматривая преобразования (два изменения цветов и два разворота), которые необходимы для выполнения вставки ключа `P` в красно-черное ДБП (см. упражнение 3.3.12).

Удаление

Поскольку метод `put()` из алгоритма 3.4 и так один из наиболее запутанных методов, которые рассматриваются в данной книге, а реализации методов `deleteMin()`, `deleteMax()` и `delete()` для красно-черных ДБП еще немного сложнее, то мы переместили их полные реализации в упражнения. Однако базовый принцип все же стоит рассмотреть. Для его описания мы вернемся к 2-3-деревьям. Как и в случае вставки, можно определить последовательность локальных преобразований, которые позволяют удалить узел, не нарушив идеальный баланс. Этот процесс несколько сложнее вставки, т.к. преобразования выполняются и при *спуске* по пути поиска, когда формируются временные 4-узлы (из которых возможно удаление), и при *подъеме* по этому же пути, когда выполняются разбиения всех возникших 4-узлов (как и при вставке).

Нисходящие 2-3-4-деревья

В качестве разминки мы рассмотрим сначала более простой алгоритм, который выполняет преобразования и при спуске, и при подъеме: алгоритм вставки для *2-3-4-деревьев*, где временные 4-узлы, которые были временным явлением в 2-3-деревьях, могут оставаться в дереве (рис. 3.3.25). Алгоритм вставки основан на преобразованиях при спуске по пути, которые сохраняют инвариант, что текущий узел не является 4-узлом (т.е. гарантированно имеется место для вставки нового ключа на нижнем уровне), и преобразованиях при подъеме, балансирующих все 4-узлы, которые могли возникнуть. Преобразования при спуске *в точности* совпадают с преобразованиями, которые применялись для разбиения 4-узлов и 2-3-деревьях. Если корень является 4-узлом, он разбивается на три 2-узла, и высота дерева увеличивается на 1. Если при спуске по дереву обнаруживается 4-узел с родительским 2-узлом, то 4-узел разбивается на два 2-узла, и средний ключ передается родителю; а если встречается 4-узел с родительским 3-узлом, то 4-узел разбивается на два 2-узла, и средний ключ передается родителю, который становится 4-узлом. Поддерживаемый инвариант позволяет не беспокоиться о случае 4-узла с родительским 4-узлом. В силу этого же инварианта на нижнем уровне возможен только 2-узел или 3-узел, т.е. место для вставки нового ключа есть.

Чтобы реализовать этот алгоритм с помощью красно-черных ДБП, нужно:

- представлять 4-узлы в виде сбалансированного поддерева трех 2-узлов, где левый и правый дочерние узлы соединяются с родителем красными связями;
- разбивать 4-узлы при *спуске* по дереву с изменением цветов;
- балансировать 4-узлы при *подъеме* по дереву с разворотами, как в случае вставки.

Интересно, что для реализации нисходящих 2-3-4-деревьев достаточно переместить одну строку кода в методе `put()` из алгоритма 3.4: а именно — вызов `colorFlip()` (и соответствующую проверку) нужно сдвинуть перед рекурсивными вызовами (между проверкой на нулевое значение и сравнением).

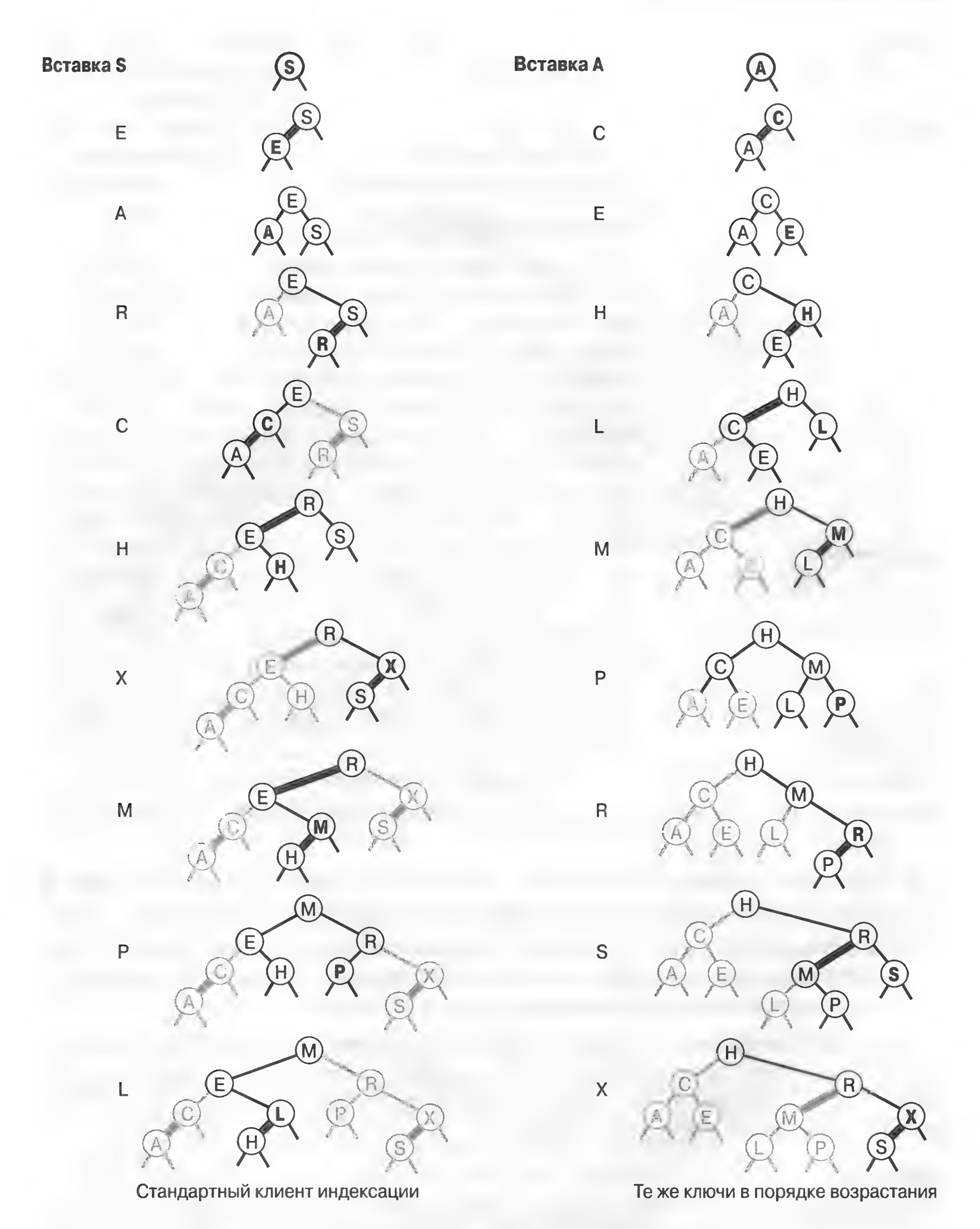


Рис. 3.3.24. Трассировки построения красно-черных ДБП

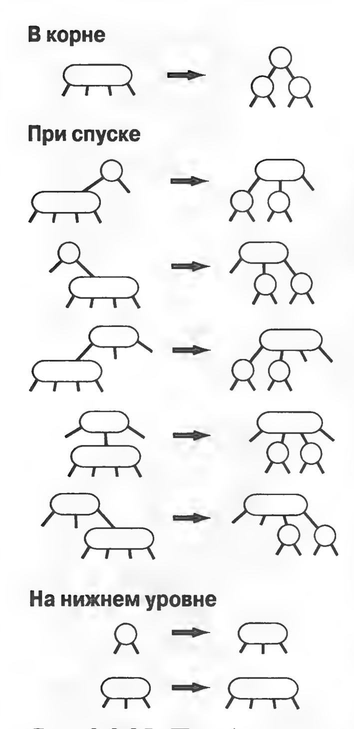


Рис. 3.3.25. Преобразования для вставки в нисходящие 2-3-4-деревья

Этот алгоритм ведет себя чуть лучше с 2-3-деревьями в приложениях, где несколько процессов имеют доступ к одному и тому же дереву, т.к. он всегда работает с одной или двумя ссылками текущего узла. Описываемые ниже алгоритмы удаления основаны на аналогичной схеме и пригодны как для таких деревьев, так и для 2-3-деревьев.

Удаление наименьшего

В качестве еще одной разминки перед реализацией удаления мы рассмотрим операцию удаления наименьшего ключа из 2-3-дерева (рис. 3.3.26). Эта операция основана на наблюдении, что ключ легко удалить из 3-узла на нижнем уровне, но не из 2-узла. После удаления ключа из 2-узла в этом узле не останется ключей. Было бы естественно заменить его просто нулевой ссылкой, но это нарушит условие идеального баланса. Поэтому мы будем придерживаться следующего подхода: чтобы не упереться внизу в 2-узел, мы выполняем соответствующие преобразования при спуске по дереву, которые поддерживают инвариант, что текущий узел не является 2-узлом (может быть 3-узлом или временным 4-узлом). Вначале в корне имеются две возможности: если корень является 2-узлом, и оба его дочерних узла тоже 2-узлы, то эти три узла просто преобразуются в 4-узел, иначе при необходимости можно выполнить заем из правого узла, чтобы левый дочерний узел не был 2-узлом.

Тогда при спуске по дереву должны встречаться описанные ниже варианты.

- Если левый дочерний узел текущего узла не является 2-узлом, то ничего не надо делать.
- Если левый дочерний узел является 2-узлом, и его непосредственный сосед не является 2-узлом, то ключ из правого дочернего узла переносится в левый.
- Если и левый дочерний узел, и его непосредственный сосед являются 2-узлами, то они объединяются в 4-узел с наименьшим ключом в родителе, а родительский узел меняется с 3-узла на 2-узел или с 4-узла на 3-узел.

Выполняя этот процесс, мы спускаемся по левым связям до самого низа и упираемся в 3-узел или 4-узел с наименьшим ключом, поэтому его можно сразу удалить, преобразовав 3-узел в 2-узел или 4-узел в 3-узел. А потом при подъеме по дереву мы разбиваем все неиспользованные временные 4-узлы.

Удаление

Те же преобразования на пути поиска, только что описанные для удаления наименьшего ключа, позволяют гарантировать, что текущий узел не является 2-узлом, и при поиске произвольного ключа. Если искомый ключ находится на нижнем уровне, его можно просто удалить. Если он находится выше, то его необходимо обменять с его последователем, как и в обычных ДБП. Потом, т.к. текущий узел не является 2-узлом, задача сводится к удалению наименьшего ключа в поддереве, корень которого не яв-

ляется 2-узлом, и можно воспользоваться описанной выше процедурой для этого поддерева. После удаления мы, как обычно, разбиваем все оставшиеся 4-узлы, поднимаясь по дереву по пути поиска.

Несколько упражнений в конце данного раздела посвящены примерам и реализациям, относящимся к этим алгоритмам удаления. Тем, кому интересны разработка и понимание реализаций, полезно будет разобраться в деталях, которым посвящены эти упражнения. А те, кто питает к изучению алгоритмов лишь общий интерес, должны понимать всю важность этих методов, т.к. они представляют первую реализацию таблицы имен из рассмотренных нами, где операции *поиска*, *вставки* и *удаления*, как мы увидим, гарантированно эффективны.

В корне

При спуске

На нижнем уровне

Рис. 3.3.26. *Преобразования для удаления минимального узла*

Свойства красно-черных деревьев

Изучение свойств красно-черных ДБП сводится к проверке соответствия с 2-3-деревьями и применения анализа к 2-3-деревьям. Конечным результатом является вывод: ***все операции в таблице имен на основе красно-черных ДБП гарантированно имеют логарифмическую сложность в зависимости от размера дерева*** (кроме поиска диапазона, где требуется дополнительное время, пропорциональное количеству возвращаемых ключей). Мы еще раз обращаем внимание на этот момент в силу его важности.

Анализ

Вначале мы покажем, что красно-черные ДБП не обязательно идеально сбалансированы, но всегда почти идеально сбалансированы, независимо от порядка вставки ключей. Этот факт непосредственно следует из взаимно однозначного соответствия с 2-3-деревьями и определяющего свойства 2-3-деревьев (идеальный баланс).

Утверждение Ж. Высота красно-черного ДБП с N узлами не превышает $2\lg N$.

Набросок доказательства. В худшем случае 2-3-дерево состоит исключительно из 2-узлов, кроме самого левого пути, который содержит только 3-узлы. Путь, проходящий по левым связям от самого корня, вдвое длиннее путей длины $\sim\lg N$, содержащий только 2-узлы. Возможно, хотя и не очень легко, придумать такие последовательности ключей, которые приводят к построению красно-черных ДБП, где средняя длина пути равна худшему случаю $2\lg N$. При склонности к математическим выкладкам вы можете самостоятельно исследовать этот вопрос, проработав упражнение 3.3.24.

Эта верхняя граница чересчур осторожная: эксперименты и со случайными последовательностями, и с последовательностями, характерными для типичных приложений,

подтверждают гипотезу, что каждый поиск в красно-черном ДБП из N узлов использует в среднем примерно 1,00 $\lg N - 0{,}5$ сравнений. Более того, на практике трудно получить значительно бóльшее среднее количество сравнений.

Свойство З. Средняя длина пути от корня до узла в красно-черном ДБП с N узлами равна ~1,00 $\lg N$.

Обоснование. Типичные деревья, вроде приведенного на рис. 3.3.27 (и даже построенные из возрастающих ключей — см. рис. 3.3.28), довольно хорошо сбалансированы по сравнению с типичными ДБП (вроде дерева, приведенного на рис. 3.2.8). В табл. 3.3.1 видно, что длины путей (стоимость поиска) для нашего приложения `FrequencyCounter` примерно на 40% короче, чем для элементарных ДБП — как и ожидалось. Этот прирост производительности наблюдался в бесчисленных приложениях и экспериментах с момента изобретения красно-черных ДБП.

Для нашего демонстрационного примера с замером стоимости операций `put()` для программы `FrequencyCounter`, работающей со словами длиной 8 символов и более, наблюдается еще большее снижение средней стоимости — т.е. подтверждение логарифмической производительности, предсказанной теоретической моделью. Правда, это подтверждение не так неожиданно, как для ДБП, в силу гарантии, сформулированной в утверждении Ж. Общая экономия не превышает 40% от стоимости поиска, т.к. мы считаем не только сравнения, но и развороты и изменения цвета.

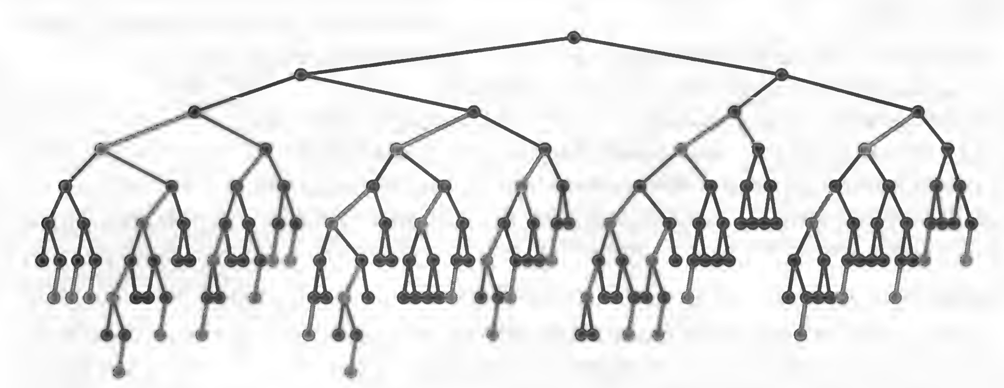


Рис. 3.3.27. Типичное красно-черное ДБП, построенное из случайных ключей (нулевые ссылки не показаны)

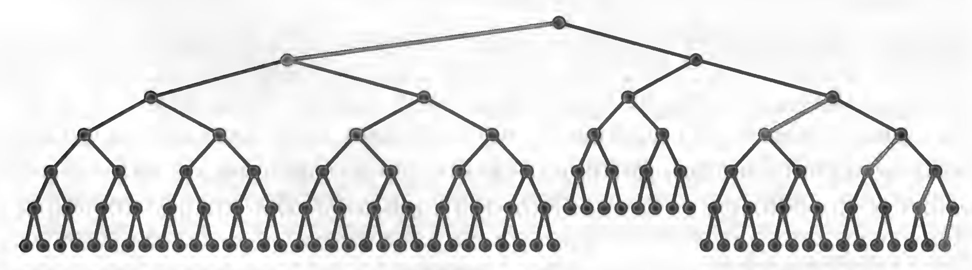


Рис. 3.3.28. Красно-черное ДБП, построенное из убывающих ключей (нулевые ссылки не показаны)

Таблица 3.3.1. Среднее количество сравнений на операцию `put()` в программе `FrequencyCounter` с использованием `RedBlackBST` (рис. 3.3.29)

	tale.txt				leipzig1M.txt			
	Слова	Различные	Сравнения		Слова	Различные	Сравнения	
			прогноз	реально			прогноз	реально
Все слова	135 635	10 679	13.6	13.5	21 191 455	534 580	19.4	19.1
Слова ≥ 8	14 350	5 737	12.6	12.1	4 239 597	299 593	18.7	18.4
Слова ≥ 10	4 582	2 260	11.4	11.5	1 610 829	165 555	17.5	17.3

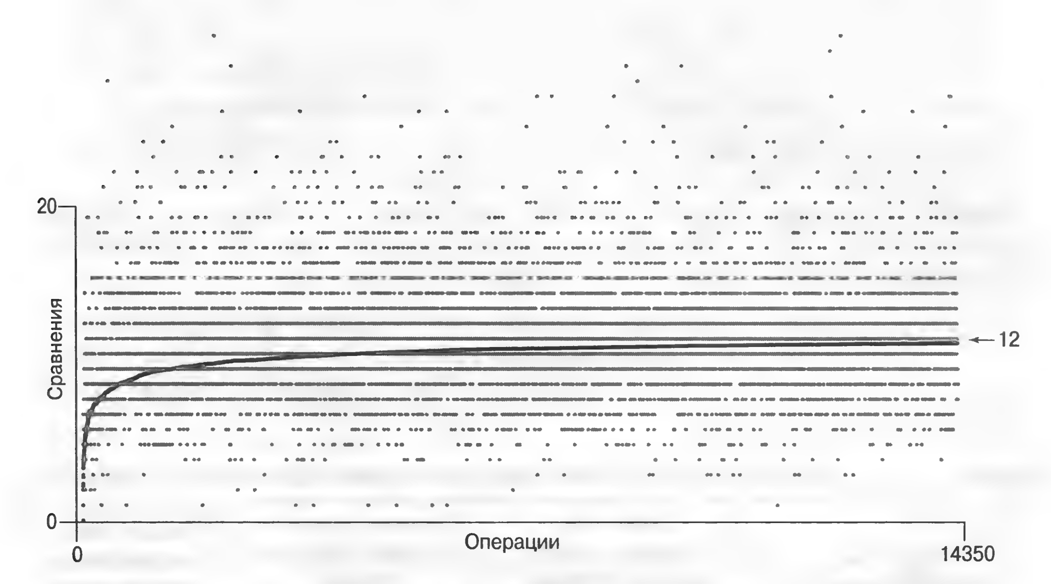


Рис. 3.3.29. Стоимости выполнения команды `java FrequencyCounter 8 < tale.txt` *с использованием* `RedBlackBST`

Метод `get()` в красно-черных ДБП не проверяет цвета узлов, поэтому механизм балансировки не увеличивает трудоемкость; поиск выполняется быстрее, чем в элементарных ДБП, в силу балансировки деревьев. Каждый ключ вставляется лишь один раз, но может принимать участие в очень многих операциях поиска; поэтому в конечном итоге значения времени поиска близки к оптимальным (т.к. деревья почти сбалансированы, а при поиске балансировка не выполняется) за счет относительно небольших усложнений (в отличие от бинарного поиска, вставки гарантированно выполняются за логарифмическое время). Внутренний цикл поиска содержит сравнение с последующим изменением ссылки — т.е. он довольно короткий, как и внутренний цикл бинарного поиска (сравнения и операции с индексами). Эта реализация — первая из описанных здесь, которая гарантирует логарифмическую сложность и поиска, и вставки, и она содержит лаконичный внутренний цикл, поэтому ее стоит применять во множестве различных приложений, включая и библиотечные.

API упорядоченной таблицы имен

Одна из наиболее привлекательных черт красно-черных ДБП состоит в том, что сложная часть кода ограничена методами вставки и удаления. Наш код для получения наименьшего и наибольшего ключа, выбора, ранга, нижней и верхней опор и запроса диапазона для стандартных ДБП можно использовать *безо всяких изменений*, т.к. он работает с ДБП и никак не учитывает цвет узлов. Алгоритм 3.4 вместе с этими методами (и методами удаления) дает полную реализацию нашего API упорядоченной таблицы имен. Более того, все эти методы выигрывают от почти идеальной балансировки дерева, поскольку всем им нужно время, не более чем пропорциональное высоте дерева. Поэтому утверждения Ж в сочетании с утверждением Д достаточно для гарантирования логарифмической производительности *всех* операций.

Утверждение И. В красно-черном ДБП следующие операции требуют логарифмического времени в худшем случае: поиск, вставка, извлечение наименьшего, извлечение наибольшего, нижняя опора, верхняя опора, ранг, выбор, удаление наименьшего, удаление наибольшего, удаление и выборка диапазона.

Доказательство. Операции `get()`, `put()` и удаление были рассмотрены выше. Для остальных операций можно использовать код из раздела 3.2 *дословно*, т.к. он игнорирует цвет узлов. Гарантия логарифмической производительности следует из утверждений Д и Ж и того факта, что каждый алгоритм выполняет постоянное количество действий с каждым просматриваемым узлом.

Вообще-то здорово то, что можно получить такие гарантии. В мире, который буквально утопает в информации, где обрабатываются таблицы с триллионами и квадриллионами записей, мы можем выполнить любую из таких операций с помощью лишь нескольких десятков сравнений.

Таблица 3.3.2. Трудоемкости для элементарных реализаций таблицы имен (обновлена)

Алгоритм (структура данных)	Стоимость в худшем случае (после *N* вставок)		Средняя стоимость (после *N* случайных вставок)		Эффективно поддерживает упорядоченные операции?
	поиск	вставка	попадание	вставка	
Последовательный поиск (неупорядоченный связный список)	N	N	$N/2$	N	нет
Бинарный поиск (упорядоченный массив)	$\lg N$	N	$\lg N$	$N/2$	да
Дерево бинарного поиска (ДБП)	N	N	$1{,}39 \lg N$	$1{,}39 \lg N$	да
Поиск в 2-3-дереве (красно-черное ДБП)	$2 \lg N$	$2 \lg N$	$1{,}00 \lg N$	$1{,}00 \lg N$	да

Вопросы и ответы

Вопрос. **Почему бы не разрешить наклон 3-узлов в любую сторону и наличие 4-узлов в дереве?**

Ответ. Это приемлемые варианты, которые также использовались десятки лет. Некоторые из них исследуются в упражнениях. Соглашение о наклоне влево сокращает количество возможных вариантов и, следовательно, требует значительно более короткого кода.

Вопрос. **Почему для представления 2-, 3- и 4-узлов не используется единый тип `Node`, в котором применяется массив значений `Key`?**

Ответ. Хороший вопрос. Именно так мы и сделаем в В-деревьях (см. главу 6), где узел может содержать значительно больше ключей. Но для небольших узлов в 2-3-деревьях затраты, необходимые для работы с массивом, слишком тяжело ложатся на производительность.

Вопрос. **При разбиении 4-узла иногда в методе `rotateRight()` правому узлу присваивается цвет `RED`, а потом тут же в методе `flipColors()` он меняется на `BLACK`. Но ведь это излишние затраты?**

Ответ. Да, иногда выполняются ненужные перекрашивания среднего узла. Но в общей картине переустановка нескольких лишних битов несопоставима с улучшением производительности с линейной до логарифмической для всех операций. Однако в приложениях, где производительность очень важна, код методов `rotateRight()` и `flipColors()` можно не выносить в отдельные конструкции и устранить лишние проверки. Эти методы используются и при удалении, и они немного легче для использования, понимания и сопровождения при сохранении идеального черного баланса.

Упражнения

3.3.1. Нарисуйте 2-3-дерево, которое получится после вставки ключей `E A S Y Q U T I O N` в указанном порядке в первоначально пустое дерево.

3.3.2. Нарисуйте 2-3-дерево, которое получится после вставки ключей `Y L P M X H C R A E S` в указанном порядке в первоначально пустое дерево.

3.3.3. Найдите такой порядок вставки ключей `S E A R C H X M`, который приводит к построению 2-3-дерева высотой 1.

3.3.4. Докажите, что высота 2-3-дерева с N ключами находится в пределах ~ $\lfloor \log_3 N \rfloor \approx 0.63 \lg N$ (для дерева, состоящего только из 3-узлов) и ~ $\lfloor \lg N \rfloor$ (для дерева, состоящего только из 2-узлов).

3.3.5. На рис. 3.3.30 приведены все *структурно различные* 2-3-деревья с N ключами для N от 1 до 6 (порядок поддеревьев игнорируется). Нарисуйте все структурно различные 2-3-деревья для N = 7, 8, 9 и 10.

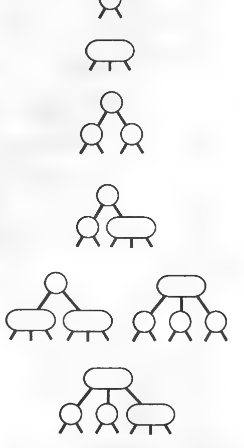


Рис. 3.3.30. *Рисунок к упражнению 3.3.5*

3.3.6. Найдите вероятность, что каждое из 2-3-деревьев из упражнения 3.3.5 является результатом вставки N случайных различных ключей в первоначально пустое дерево.

3.3.7. Начертите диаграммы наподобие рис. 3.3.9 для остальных пяти случаев, представленных на рис. 3.3.8.

3.3.8. Приведите все возможные способы представления 4-узла с помощью трех 2-узлов, соединенных красными связями (не обязательно направленных влево).

3.3.9. Какие деревья из приведенных на рис. 3.3.31 являются красно-черными ДБП?

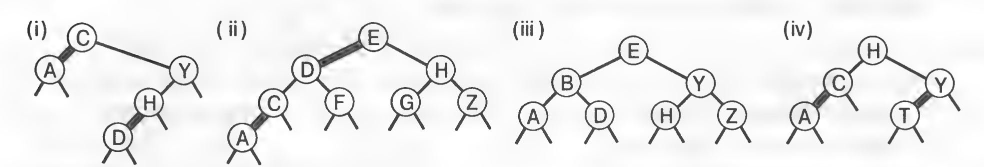


Рис. 3.3.31. Рисунок к упражнению 3.3.9

3.3.10. Нарисуйте красно-черное ДБП, которое получится после вставки ключей `E A S Y Q U T I O N` в указанном порядке в первоначально пустое дерево.

3.3.11. Нарисуйте красно-черное ДБП, которое получится после вставки ключей `Y L P M X H C R A E S` в указанном порядке в первоначально пустое дерево.

3.3.12. Нарисуйте красно-черное ДБП, которое получается после каждого преобразования (изменения цвета или разворота) при вставке ключа `P` в нашем стандартном клиенте индексации.

3.3.13. Правда ли, что при вставке ключей в порядке возрастания в красно-черное ДБП высота этого дерева монотонно возрастает?

3.3.14. Нарисуйте красно-черное ДБП, которое получится при вставке букв от `A` до `K` по порядку в первоначально пустое дерево. После этого опишите, что происходит в общем случае при построении деревьев вставками ключей в порядке возрастания (см. также рисунок в тексте).

3.3.15. Ответьте на два предыдущих вопроса для случая, когда ключи вставляются в порядке *убывания*.

3.3.16. Приведите результат вставки ключа `n` в красно-черное ДБП на рис. 3.3.32 (показан только путь поиска, и в ответе нужно привести только эти узлы).

3.3.17. Сгенерируйте два случайных красно-черных ДБП из 16 узлов. Начертите их (программно или от руки). Сравните их с (несбалансированными) ДБП, построенными из тех же ключей.

3.3.18. Нарисуйте все структурно различные красно-черные ДБП из N ключей, для N от 2 до 10 (см. упражнение 3.3.5).

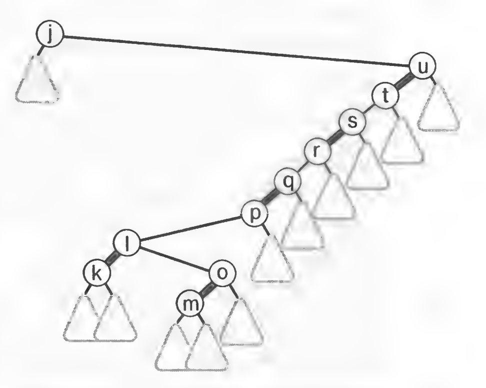


Рис. 3.3.32. Рисунок к упражнению 3.3.16

3.3.19. Один бит цвета на узел позволяет представлять 2-, 3- и 4-узлы. Сколько битов цвета должен содержать узел для представления 5-, 6-, 7- и 8-узлов в бинарном дереве?

3.3.20. Вычислите длину внутреннего пути в идеально сбалансированном ДБП из N узлов, если N равно степени двух минус 1.

3.3.21. Напишите клиент тестирования `TestRB.java`, воспользовавшись решением упражнения 3.2.10.

3.3.22. Найдите последовательность ключей, которую надо вставить в обычное и красно-черное ДБП, чтобы высота обычного ДБП была меньше высоты красно-черного ДБП, или докажите, что такая последовательность невозможна.

Творческие задачи

3.3.23. *2-3-деревья без требования баланса.* Разработайте реализацию базового API таблицы имен на основе 2-3-деревьев, которые не обязательно сбалансированы. Разрешите наклон 3-узлов в любую сторону. При вставке в 3-узел на нижнем уровне прикрепляйте новый узел *черной* связью. Проведите эксперименты и на их основе сформулируйте гипотезу с оценкой средней длины пути в дереве, построенном N случайными вставками.

3.3.24. *Худший случай для красно-черных ДБП.* Объясните, как нужно построить красно-черное ДБП, демонстрирующее, что в худшем случае почти все пути от корня до нулевой ссылки в красно-черном ДБП из N узлов имеют длину $2\lg N$.

3.3.25. *Нисходящие 2-3-4 деревья.* Разработайте реализацию базового API таблицы имен на основе сбалансированных 2-3-4-деревьев, красно-черного представления и метода вставки, описанного в тексте. 4-узлы должны разбиваться с помощью изменения цвета при спуске по пути поиска и балансироваться при подъеме по этому пути.

3.3.26. *Единый нисходящий проход.* Разработайте вариант решения упражнения 3.3.25, в котором *не* применяется рекурсия. Выполняйте все разбиения и балансировку 4-узлов (и 3-узлов тоже) при спуске по дереву, который завершается вставкой ключа на нижнем уровне.

3.3.27. *Разрешение правых красных связей.* Разработайте вариант решения упражнения 3.3.25, в котором разрешены правые красные связи в дереве.

3.3.28. *Восходящие 2-3-4-деревья.* Разработайте реализацию базового API таблицы имен на основе сбалансированных 2-3-4-деревьев, красно-черного представления и *восходящего* метода вставки, где используется тот же рекурсивный подход, что и в алгоритме 3.4. Метод вставки должен разбивать только последовательность 4-узлов (если они есть) в нижней части пути поиска.

3.3.29. *Оптимальный объем памяти.* Измените класс `RedBlackBST`, чтобы не использовать дополнительную память для бита цвета, с помощью следующего приема: чтобы закрасить узел красным цветом, обменяйте его ссылки. Тогда для проверки, что узел красный, достаточно проверить, что его левый дочерний узел больше правого. При этом потребуется изменить сравнения, чтобы они учитывали возможный обмен ссылок, и этот прием заменяет сравнение бита сравнением

ключей, что может быть сложнее. Но он демонстрирует, что при необходимости бит цвета можно убрать из узлов.

3.3.30. *Программное кеширование.* Добавьте в класс `RedBlackBST` хранение последнего узла, к которому было обращение, в переменной экземпляров, чтобы иметь доступ к этому узлу за постоянное время, если следующая операция `put()` или `get()` использует тот же ключ (см. упражнение 3.1.25).

3.3.31. *Вычерчивание дерева.* Добавьте в класс `RedBlackBST` метод `draw()`, который вычерчивает красно-черные ДБП в том же стиле, который применяется в тексте (см. упражнение 3.2.38).

3.3.32. *АВЛ-деревья. АВЛ-дерево* (названное по первым буквам фамилий создателей — Г.М. Адельсон-Вельского и Е.М. Ландиса — *прим. перев.*) представляет собой ДБП, в котором высота каждого узла отличается от высоты его родственного узла не более чем на 1. (Самые старые алгоритмы работы со сбалансированными деревьями основаны на разворотах, поддерживающих баланс высот в АВЛ-деревьях.) Покажите, что закраска красным цветом связей, которые ведут от узлов с четной высотой к узлам с нечетной высотой в АВЛ-деревьях, дает (идеально сбалансированное) 2-3-4-дерево, где красные связи не обязательно направлены влево. *Дополнительное задание*: разработайте реализацию API таблицы имен, основанной на этой структуре данных. Один из способов — применение поля высоты в каждом узле и выполнение разворотов после рекурсивных вызовов для возможной коррекции высоты; другой способ — использование красно-черного представления и методов наподобие `moveRedLeft()` и `moveRedRight()` из упражнений 3.3.39 и 3.3.40.

3.3.33. *Проверка.* Добавьте в класс `RedBlackBST` метод `is23()` для выполнения проверки, что ни у одного узла нет двух красных ссылок и нет правых красных ссылок, и метод `isBalanced()` для выполнения проверки, что все пути от корня к нулевым ссылкам содержат одинаковое количество черных ссылок. Объедините эти методы с кодом `isBST()` из упражнения 3.2.32, чтобы получить метод `isRedBlackBST()`, который проверяет, что дерево является красно-черным деревом бинарного поиска.

3.3.34. *Все 2-3-деревья.* Напишите программу для генерации всех структурно различных 2-3-деревьев высотой 2, 3 и 4. Должно получиться 2, 7 и 122 таких деревьев соответственно. (*Совет*: воспользуйтесь таблицей имен.)

3.3.35. *2-3-деревья.* Напишите программу `TwoThreeST.java`, которая использует два типа узлов для непосредственной реализации 2-3-деревьев поиска.

3.3.36. *2-3-4-5-6-7-8-деревья.* Опишите алгоритмы для поиска и вставки в сбалансированные 2-3-4-5-6-7-8-деревья поиска.

3.3.37. *Эффект памяти.* Покажите, что красно-черные ДБП *обладают памятью*: например, если вставить в него ключ, меньший всех других ключей, и сразу же удалить наименьший ключ, то может получиться другое дерево.

3.3.38. *Фундаментальная теорема о разворотах.* Покажите, что любое ДБП можно преобразовать в любое другое ДБП из того же множества ключей последовательностью разворотов влево и вправо.

3.3.39. *Удаление наименьшего.* Реализуйте операцию `deleteMin()` для красно-черных ДБП, используя преобразования, описанные в тексте, для спуска по левому пути дерева и поддерживая инвариант, что текущий узел не является 2-узлом.

Решение:

```
private Node moveRedLeft(Node h)
{  // Если узел h красный, а обе ссылки h.left и h.left.left черные,
   // то закрашиваем h.left или один из его потомков красным.
   flipColors(h);
   if (isRed(h.right.left))
   {
      h.right = rotateRight(h.right);
      h = rotateLeft(h);
   }
   return h;
}

public void deleteMin()
{
   if (!isRed(root.left) && !isRed(root.right))
      root.color = RED;
   root = deleteMin(root);
   if (!isEmpty()) root.color = BLACK;
}

private Node deleteMin(Node h)
{
   if (h.left == null)
      return null;
   if (!isRed(h.left) && !isRed(h.left.left))
      h = moveRedLeft(h);
   h.left = deleteMin(h.left);
   return balance(h);
}
```

В этом коде используется метод `balance()`, состоящий из строки кода

```
if (isRed(h.right)) h = rotateLeft(h);
```

за которой следуют последние пять строк из рекурсивного метода `put()` из алгоритма 3.4, и реализация `flipColors()`, которая выполняет дополнение трех цветов (вместо метода вставки, приведенного в тексте).

3.3.40. *Удаление наибольшего.* Реализуйте операцию `deleteMax()` для красно-черных ДБП. Учтите, что необходимые для этого преобразования несколько отличаются от преобразований из предыдущего упражнения, т.к. красные ссылки направлены влево.

Решение:

```
private Node moveRedRight(Node h)
{  // Если узел h красный, а обе ссылки h.right и h.right.left черные,
   // то закрашиваем h.right или один из его потомков красным.
   flipColors(h)
   if (!isRed(h.left.left))
      h = rotateRight(h);
   return h;
}
```

```
public void deleteMax()
{
  if (!isRed(root.left) && !isRed(root.right))
    root.color = RED;
  root = deleteMax(root);
  if (!isEmpty()) root.color = BLACK;
}
private Node deleteMax(Node h)
{
  if (isRed(h.left))
     h = rotateRight(h);
  if (h.right == null)
    return null;
  if (!isRed(h.right) && !isRed(h.right.left))
    h = moveRedRight(h);
  h.right = deleteMax(h.right);
  return balance(h);
}
```

3.3.41. *Удаление.* Реализуйте операцию `delete()` для красно-черных ДБП, объединив методы из двух предыдущих упражнений с операцией `delete()` для обычных ДБП.

***Решение*:**

```
public void delete(Key key)
{
  if (!isRed(root.left) && !isRed(root.right))
    root.color = RED;
  root = delete(root, key);
  if (!isEmpty()) root.color = BLACK;
}

private Node delete(Node h, Key key)
{
  if (key.compareTo(h.key) < 0)
  {
    if (!isRed(h.left) && !isRed(h.left.left))
      h = moveRedLeft(h);
    h.left =  delete(h.left, key);
  }
  else
  {
    if (isRed(h.left))
      h = rotateRight(h);
    if (key.compareTo(h.key) == 0 && (h.right == null))
      return null;
    if (!isRed(h.right) && !isRed(h.right.left))
      h = moveRedRight(h);
    if (key.compareTo(h.key) == 0)
    {
      h.val = get(h.right, min(h.right).key);
      h.key = min(h.right).key;
      h.right = deleteMin(h.right);
    }
    else h.right = delete(h.right, key);
  }
  return balance(h);
}
```

Эксперименты

3.3.42. *Подсчет красных узлов.* Напишите программу, которая вычисляет процент красных узлов в заданном красно-черном ДБП. Проверьте работу программы, выполнив, по крайней мере, 100 раз вставку N случайных ключей в первоначально пустое дерево, для $N = 10^4$, 10^5 и 10^6, и сформулируйте гипотезу.

3.3.43. *Графики стоимости.* Добавьте в класс `RedBlackBST` средства вывода графиков наподобие приведенных в данном разделе, которые показывают стоимость одной операции `put()` (см. упражнение 3.1.38).

3.3.44. *Среднее время поиска.* Эмпирически определите среднее значение и среднеквадратичное отклонение для длины пути к случайному узлу (длина внутреннего пути, деленная на размер дерева) в красно-черном ДБП, построенном вставками N случайных ключей в первоначально пустое дерево для N от 1 до 10000. Выполните, по крайней мере, 1000 построений для каждого размера дерева. Результаты оформите в виде графика Тафти (Tufte), вроде показанного на рис. 3.3.33, и аппроксимируйте график функцией $\lg N - 0{,}5$.

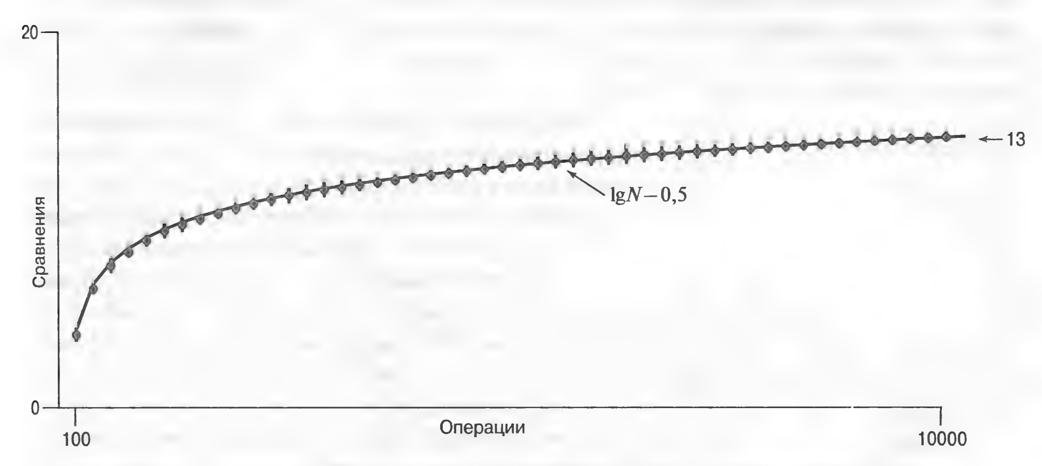


***Рис. 3.3.33.** Средняя длина пути к случайному узлу в красно-черном ДБП, построенном из случайных ключей*

3.3.45. *Подсчет разворотов.* Добавьте в программу из упражнения 3.3.43 средство создания графика для количества разворотов и разбиений узлов, которые выполняются при построении деревьев. Проанализируйте результаты.

3.3.46. *Высота.* Добавьте в программу из упражнения 3.3.43 создание графика высоты красно-черных ДБП. Проанализируйте результаты.

3.4. Хеш-таблицы

Если ключи представляют собой небольшие целые числа, то для реализации неупорядоченной таблицы имен можно использовать массив, рассматривая ключи как индексы этого массива. Тогда значение, связанное с ключом `i`, можно сохранить в элементе массива `i`, и оно будет готово для непосредственного доступа. В данном разделе мы рассмотрим *хеширование* — расширение этого простого метода, которое пригодно для более сложных видов ключей. Мы обращаемся к парам ключ-значение с помощью массивов и арифметических операций для преобразования ключей в индексы массивов.

Алгоритмы поиска, использующие хеширование, состоят из двух отдельных частей. Первая часть — вычисление *хеш-функции*, которая преобразует искомый ключ в индекс массива. В идеале различные ключи должны отображаться на различные индексы. Этот идеал обычно недостижим, поэтому приходится учитывать возможность, что два или более различных ключей могут преобразоваться в один и тот же индекс массива (рис. 3.4.1). Поэтому вторая часть хеширующего поиска — это процесс *разрешения коллизий*, который как раз и обрабатывает такие ситуации. После описания способов вычисления хеш-функций мы рассмотрим два различных способа разрешения коллизий: *раздельные цепочки* и *линейное опробование*.

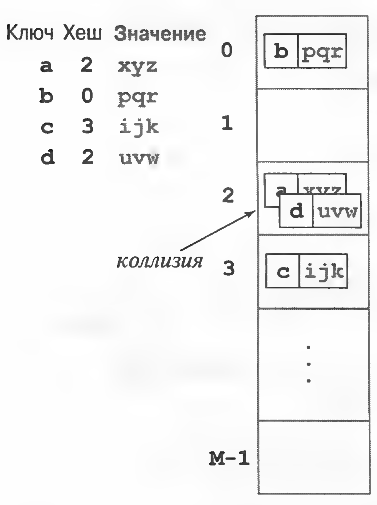


Рис. 3.4.1. *Хеширование: суть проблемы*

Хеширование представляет собой типичный пример *компромисса между временем и памятью*. При отсутствии каких-либо ограничений на память любой поиск можно выполнить, просто обратившись к памяти с ключом в качестве индекса в (потенциально огромном) массиве. Однако обычно этот идеал недостижим, т.к. при очень большом количестве возможных ключей объем необходимой памяти неподъемно велик. А если нет ограничения на время, можно задействовать лишь минимальное количество памяти, выполнив последовательный поиск в неупорядоченном массиве. Хеширование — это способ использовать разумное количество как памяти, так и времени и найти баланс между этими двумя крайностями. Такой компромисс между временем и памятью в алгоритмах хеширования можно варьировать, настраивая параметры и не переписывая код. А для более осознанного выбора значений этих параметров мы используем классические результаты из теории вероятностей.

Теория вероятностей — триумф математического анализа, но она выходит за рамки настоящей книги. Однако алгоритмы хеширования, которые мы будем рассматривать, довольно просты и широко применимы, хотя и основаны на этой теории. Хеширование позволяет реализовать поиск и вставку в таблицах имен, которые требуют *константного* (в амортизированном смысле) времени на одну операцию в типичных приложениях, что делает его рекомендуемым способом для реализации базовых таблиц имен во многих ситуациях.

Хеш-функции

Первая наша задача — вычисление хеш-функции, которая преобразует ключи в индексы массива. Если имеется массив, который может вместить M пар ключ-значение, то *хеш-функция* должна преобразовывать любой заданный ключ в индекс в этом массиве, т.е. целое число из диапазона $[0, M-1]$. Нам нужна такая хеш-функция, которая легко вычисляется и равномерно распределяет ключи: для любого ключа одинаково вероятным должен быть целочисленный результат от 0 до $M-1$ (независимо для каждого ключа). Этот идеал несколько загадочен, и для понимания хеширования стоит внимательно разобраться, как же реализовать такую функцию.

Хеш-функция зависит от типа ключа. Строго говоря, *для каждого типа ключа нужна особая хеш-функция*. Если ключ содержит число, вроде идентификационного кода, можно начать с этого номера; если в ключе имеется строка, подобная имени человека, необходимо преобразовать эту строку в число; а если в ключе есть несколько частей, как в почтовом адресе, но все эти части нужно как-то объединить. Для многих распространенных типов ключей можно использовать стандартные реализации, уже имеющиеся в Java. Сейчас мы кратко рассмотрим возможные реализации для различных видов ключей, чтобы было понятно, с чем нам придется иметь дело, т.к. в дальнейшем вам понадобится выполнять реализации для типов ключей, которые вы будете создавать сами.

Типичный пример

Допустим, имеется приложение, где в качестве ключей выступают номера карточек социального страхования, принятые в США. Такой номер имеет вид `123-45-6789` и состоит из девяти цифр, сгруппированных в три поля. Первое поле означает географический регион, в котором был выдан номер (например, номера с цифрами `035` в первом поле выданы в Род-Айленде, а с цифрами `214` — в Мэриленде). Два других поля индивидуальны. Всего может быть миллиард (10^9) различных номеров карточек социального страхования, но предположим, что в нашем приложении нужно обработать лишь несколько сотен ключей, и можно использовать хеш-таблицу размером $M = 1000$. Один возможный способ реализации хеш-функции — использовать три какие-то цифры из ключа. Лучше, конечно, брать три цифры из третьего поля, а не из первого: клиенты могут быть неравномерно распределены по регионам. Но еще лучше задействовать все девять цифр, чтобы получить значение `int`, а затем подобрать хеш-функцию для целочисленных значений.

Положительные целые числа

Для хеширования целочисленных ключей чаще всего применяется *модульное хеширование*: выбираем размер массива M так, чтобы он был простым числом, и для любого положительного целого ключа k вычисляем остаток от деления k на M (рис. 3.4.2). Эта функция очень легко вычисляется (в Java — `k % M`) и довольно равномерно распределяет ключи между 0 и $M-1$. Если M не простое число, то может оказаться так, что не все биты принимают участие в вычислениях, а это ухудшает рав-

Ключ	Хеш (M = 100)	Хеш (M = 97)
212	12	18
618	18	36
302	2	11
940	40	67
702	2	23
704	4	25
612	12	30
606	6	24
772	72	93
510	10	25
423	23	35
650	50	68
317	17	26
907	7	34
507	7	22
304	4	13
714	14	35
857	57	81
801	**1**	25
900	0	27
413	13	25
701	**1**	22
418	18	30
601	**1**	19

Рис. 3.4.2. *Модульное хеширование*

номерность распределения значений. Например, если ключи — десятичные числа, а M равно 10^k, то будут задействованы лишь k последних значащих цифр. В качестве простого примера, где такой выбор может оказаться неудачным, допустим, что ключи представляют собой международные телефонные коды, и $M = 100$. По историческим причинам большинство телефонных кодов в США содержат среднюю цифру 0 или 1, и такой выбор будет давать значения, меньшие 20, а использование простого значения 97 распределит их лучше (а еще лучше — простое значение, далекое от 100). По похожим историческим причинам не случайно распределены и IP-адреса, используемые в Интернете, поэтому размер таблицы лучше сделать простым (в частности, *не* равным степени 2), если требуется их равномерное распределение с помощью модульного хеширования.

Числа с плавающей точкой

Если ключи представляют собой вещественные числа от 0 до 1, то можно умножить их на M и округлить до ближайшего целого, чтобы получить индекс от 0 до $M-1$. Этот способ с виду естественен, но он придает больший вес самым значимым битам ключа, а наименее значимые биты могут вообще не играть никакой роли. Один из возможных способов исправить эту ситуацию — применение модульного хеширования для двоичного представления ключа (так и принято в Java).

Строки

Модульное хеширование пригодно и для длинных ключей наподобие строк: они просто рассматриваются как очень длинные целые числа. Например, в листинге 3.4.1 вычисляется модульная хеш-функция для аргумента `String s`: вспомните, что функция `charAt()` возвращает в Java значение `char` — 16-разрядное неотрицательное целое число. Если `R` больше значения любого символа, то это вычисление будет эквивалентно работе с объектом `String` как с `N`-значным целым числом, записанным по основанию `R`, когда вычисляется остаток от деления этого числа на `M`. Классический алгоритм — *метод Горнера* — выполняет такое вычисление с помощью `N` умножений, сложений и взятий модуля. Если `R` достаточно мало, чтобы не вызвать переполнения, результатом будет целое число от `0` до `M-1`, что как раз и требуется. Использование небольшого простого числа вроде 31 гарантирует, что в результате будут учтены биты всех символов.

Листинг 3.4.1. Хеширование строкового ключа

```
int hash = 0;
for (int i = 0; i < s.length(); i++)
   hash = (R * hash + s.charAt(i)) % M;
```

Составные ключи

Если ключ содержит несколько целочисленных полей, то их обычно можно объединить примерно так, как описано для значений `String`. Например, пусть ключ поиска имеет тип `Date`, который содержит три целочисленных поля: `day` (двузначное число месяца), `month` (двузначный номер месяца) и `year` (четырехзначный номер года). Тогда можно вычислить число

```
int hash = (((day * R + month) % M ) * R + year) % M;
```

и если значение `R` достаточно мало, чтобы не вызвать переполнения, то это будет число от `0` до `M-1`, что как раз и требуется. В данном случае стоимость внутренней операции `%M` можно снизить, выбрав для `R` не слишком большое простое значение вроде 31. Как и в случае строк, этот метод можно обобщить на любое количество полей.

Соглашения, принятые в Java

Чтобы справиться с базовой проблемой, что каждый тип данных должен иметь какую-то хеш-функцию, в Java каждый тип данных наследует метод `hashCode()`, который возвращает 32-битное целое число. Реализация этого метода для произвольного типа данных должна быть *согласована с равенством*. То есть если истинно `a.equals(b)`, то выражение `a.hashCode()` должно иметь то же числовое значение, что и `b.hashCode()`. А если значения `hashCode()` различны, то из этого следует, что объекты не равны. Если значения `hashCode()` совпадают, то объекты могут быть, а могут и не быть равны, и для выяснения ситуации необходимо применять метод `equals()`. Данное соглашение — основное требование, которому должны следовать клиенты для использования метода `hashCode()` для таблиц имен. Из этого следует, что если понадобится выполнять хеширование с пользовательским типом, то придется переопределить методы `hashCode()` и `equals()`. Стандартная реализация возвращает машинный адрес объекта ключа, но это редко бывает то, что нужно. В Java предусмотрены реализации `hashCode()`, которые переопределяют стандартные для многих часто используемых типов (в том числе `String`, `Integer`, `Double`, `File` и `URL`).

Преобразование `hashCode()` в индекс массива

Поскольку нам нужен индекс массива, а не 32-битное целое число, мы в наших реализациях объединяем вызов `hashCode()` с модульных хешированием, которое дает целые числа от 0 до `M-1`:

```
private int hash(Key x)
{  return (x.hashCode() & 0x7fffffff) % M;  }
```

Этот код маскирует знаковый бит (чтобы превратить 32-битное число в неотрицательное 31-битное), а затем, как и в модульном хешировании, вычисляет остаток от деления на `M`. При использовании такого кода обычно в качестве размера хеш-таблицы берут *простое* число — тогда задействуются все биты хеш-кода. *Примечание.* Чтобы не загромождать наши примеры, мы не будем выполнять все эти вычисления и использовать вместо них значения, приведенные на рис. 3.4.3.

Ключ	S	E	A	R	C	H	X	M	P	L
Хеш (M = 5)	2	0	0	4	4	4	2	4	3	3
Хеш (M = 16)	6	10	4	14	5	4	15	1	14	6

Рис. 3.4.3. *Хеш-значения для ключей из примеров*

Пользовательская функция `hashCode()`

Клиентский код ожидает, что функция `hashCode()` равномерно распределяет ключи на все возможные 32-битные значения результата. То есть для любого объекта `x` можно записать `x.hashCode()` и в принципе ожидать, что с одинаковой вероятностью будет возвращено одно из 2^{32} возможных 32-битных значений. Реализации `hashCode()`, включенные в Java для типов `String`, `Integer`, `Double`, `File` и `URL`, придерживаются этого соглашения, но для собственного типа вам придется сделать все самостоятельно. Рассмотренный выше пример с типом `Date` демонстрирует один из способов — перевод переменных экземпляров в целые числа и применение модульного хеширования.

Соглашение, принятое в Java, что все типы данных наследуют метод hashCode(), позволяет сделать проще: использовать метод hashCode() для переменных экземпляров, чтобы преобразовать каждое из них в 32-битное значение int, а затем выполнить арифметические преобразования, приведенные в листинге 3.4.2 для класса Transaction. Учтите, что переменные примитивных типов необходимо преобразовать в оболочечный тип — только тогда будет доступен метод hashCode(). Точные значения множителя здесь также не очень важны (в нашем примере — 31).

Листинг 3.4.2. Реализация функции hashCode() в определении пользовательского типа

```
public class Transaction
{
   ...
   private final String who;
   private final Date when;
   private final double amount;
   public int hashCode()
   {
      int hash = 17;
      hash = 31 * hash + who.hashCode();
      hash = 31 * hash + when.hashCode();
      hash = 31 * hash
         + ((Double) amount).hashCode();
      return hash;
   }
   ...
}
```

Программное кеширование

Если вычисление хеш-кода трудоемко, можно *кешировать* хеш для каждого ключа. То есть в тип ключа можно добавить переменную экземпляров hash, которая будет содержать значение hashCode() для каждого объекта ключа (см. упражнение 3.4.25). При первом вызове hashCode() значение хеша придется вычислить, но последующие вызовы будут просто возвращать значение hash. Этот прием используется в Java для снижения стоимости вычисления hashCode() для объектов String.

Итак, имеются три основных требования к реализации хорошей хеш-функции для заданного типа данных.

- Она должна быть *согласованной*: равные ключи должны давать одно и то же хеш-значение.
- Она должна *эффективно вычисляться*.
- Она должна *равномерно распределять ключи*.

Одновременное удовлетворение всех этих требований — задача для экспертов. Как и в случае многих других возможностей, встроенных в Java, программисты, использующие хеширование, считают, что функция hashCode() справляется с этой задачей, если нет свидетельств обратного.

Но все-таки следует проявлять осторожность, работая с хешированием в таких ситуациях, где хорошая производительность критична. Плохая хеш-функция представляет собой классический пример *ошибки производительности*: все работает нормально, но медленнее, чем ожидалось.

Равномерность проще всего обеспечить, сделав так, чтобы все биты ключа играли одинаковую роль в вычислении каждого хеш-значения: пожалуй, наиболее распространенная ошибка при реализации хеш-функций — игнорирование значащих битов ключа. Но при любой реализации разумно будет протестировать вашу хеш-функцию, если производительность приложения важна. Что требует больше времени — вычисление хеш-функции или сравнение двух ключей? Равномерно ли распределяет хеш-функция типичное множество ключей на значения от 0 до $M-1$? Простые эксперименты, которые ответят на эти вопросы, могут защитить в будущем клиенты от неприятных сюрпризов. Например, гистограмма на рис. 3.4.4 показывает, что наша реализация функции `hash()` на основе метода `hashCode()` для класса `String` дает приемлемое распределение слов из текста "Tale of Two Cities".

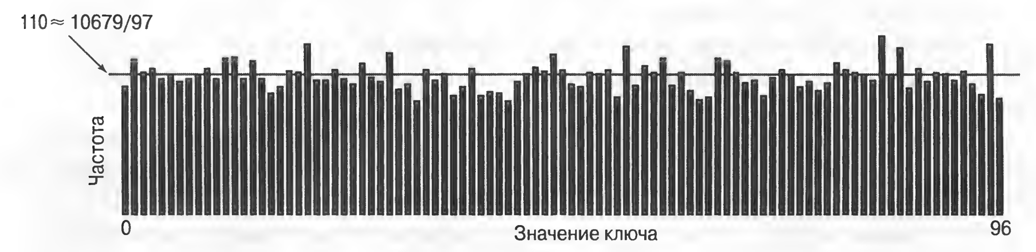


Рис. 3.4.4. Частоты хеш-значений для слов из "Tale of Two Cities" (10 679 ключей, M = 97)

Наше обсуждение основано на фундаментальном предположении, которое мы применяем при использовании хешировании. Это предположение является идеализированной моделью, которую мы не ожидаем получить реально, но которая будет направлять ход наших мыслей при реализации алгоритмов хеширования:

Предположение К (*предположение о равномерном хешировании*). Применяемые нами хеш-функции равномерно и независимо распределяют ключи на целые значения от 0 до $M-1$.

Обсуждение. Конечно, все наши произвольно выбранные параметры не могут дать равномерное и независимое распределение ключей в строгом математическом смысле. Более того, идея реализации согласованных функций, которые гарантируют равномерное и независимое распределение ключей, уводит нас в такие глубокие теоретические изыскания, что мы вряд ли сможем легко вычислять подобные функции. На практике, как и в случае случайных чисел, генерируемых функцией `Math.random()`, большинство программистов стараются получить хеш-функции, которые трудно отличить от случайных. Однако лишь немногие проверяют независимость, и это требование редко выполняется.

Несмотря на сложность проверки, предположение К является полезным способом рассматривать хеширование по двум основным причинам. Во-первых, при проектировании хеш-функции стоит держать в уме цель, которая уведет нас от произвольных решений с потенциально большим количеством коллизий. Во-вторых, хотя мы и не можем проверить само предположение, оно позволяет использовать математический анализ для разработки гипотез о производительности алгоритмов хеширования, которые затем можно проверять экспериментально.

14 Зак. 3239

Хеширование с раздельными цепочками

Хеш-функция преобразует ключи в индексы массива. Вторым компонентом алгоритма хеширования является *разрешение коллизий* — стратегия поведения в случаях, когда два или более вставляемых ключей хешируются в один и тот же индекс. Естественным и общим способом разрешения коллизий является построение для каждого из M индексов массива связного списка пар ключ-значение, ключи которых хешируются в данный индекс. Этот метод называется *раздельными цепочками* (или *цепочками переполнения — прим. перев.*), т.к. "столкнувшиеся" элементы собираются в раздельных связных списках. Главное здесь выбрать достаточно большое M, чтобы списки были не слишком длинными и не затрудняли поиск. То есть получается двухшаговый процесс: вначале находится список, который может содержать искомый ключ, а потом выполняется последовательный поиск ключа в этом списке.

Один из приемлемых способов — добавление в класс `SequentialSearchST` (алгоритм 3.1) реализации раздельных цепочек с помощью примитивов связных списков (см. упражнение 3.4.2).

Однако имеется более простой (хотя и не столь эффективный) способ: для каждого из M индексов массива строится *таблица имен* ключей, которые хешируются в данный индекс. Это позволит применять уже разработанный код.

Реализация `SeparateChainingHashST` в алгоритме 3.5 использует массив объектов `SequentialSearchST` (рис. 3.4.5), а для реализации методов `get()` и `put()` вычисляет хеш-функцию, на основе которой определяется нужный объект `SequentialSearchST`, который может содержать ключ, а затем вызывает метод `get()` или `put()` (соответственно) из класса `SequentialSearchST`, чтобы завершить задачу.

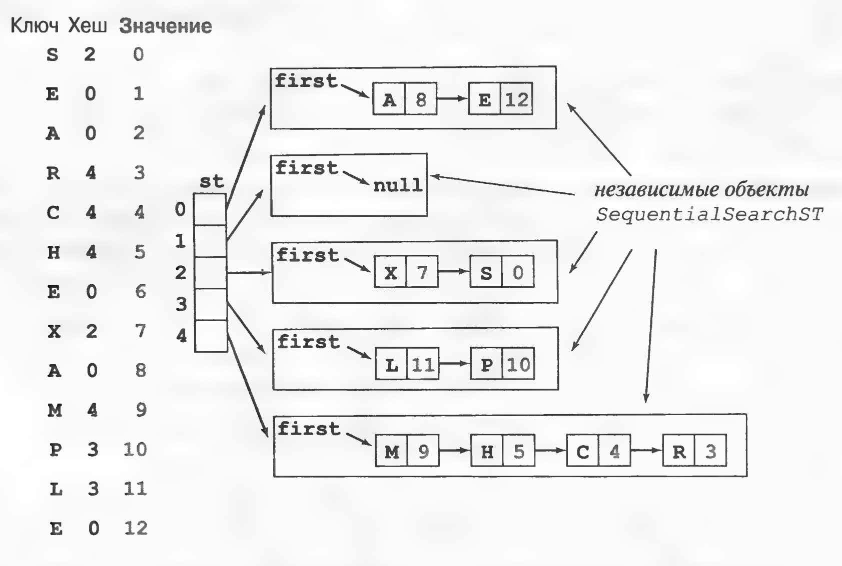


Рис. 3.4.5. Хеширование с раздельными цепочками для стандартного клиента индексации

Листинг 3.4.3. Алгоритм 3.5. Хеширование с раздельными цепочками

```
public class  SeparateChainingHashST<Key, Value>
{
  private int N;                              // количество пар ключ-значение
  private int M;                              // размер хеш-таблицы
  private SequentialSearchST<Key, Value>[] st;  // массив объектов ST

  public SeparateChainingHashST()
  {  this(997);  }

  public SeparateChainingHashST(int M)
  {  // Создание M связных списков.
     this.M = M;
     st = (SequentialSearchST<Key, Value>[]) new SequentialSearchST[M];
     for (int i = 0; i < M; i++)
        st[i] = new SequentialSearchST();
  }

  private int hash(Key key)
  {  return (key.hashCode() & 0x7fffffff) % M; }

  public Value get(Key key)
  {  return (Value) st[hash(key)].get(key);  }

  public void put(Key key, Value val)
  {  st[hash(key)].put(key, val);  }

  public Iterable<Key> keys()
  // См. упражнение 3.4.19.
}
```

В этой базовой реализации таблицы имен задействован массив связных списков, которые выбираются для каждого ключа на основе вычисления хеш-функции. Для простоты здесь используются методы класса `SequentialSearchST`. При создании массива `st[]` необходимо выполнить приведение типа, т.к. Java запрещает массивы с обобщенными типами. Стандартный конструктор задает 997 списков, поэтому для больших таблиц этот код примерно в 1000 раз быстрее `SequentialSearchST`. Данное несложное решение представляет собой легкий способ получить хорошую производительность, если имеется хоть какая-то предварительная оценка количества пар ключ-значение, которые поместит клиент с помощью метода `put()`. Более устойчивое решение — изменение размера массива, чтобы списки оставались короткими независимо от количества пар ключ-значение в таблице (см. ниже раздел "Изменение размера массива" и упражнение 3.4.18).

Если имеются M списков и N ключей, то средняя длина списков *всегда* равна N/M, независимо от распределения ключей между списками. Даже если все элементы попали в первый список, средняя длина списков равна $(N + 0 + 0 + ... + 0) / M = N/M$. *Однако* ключи распределены по спискам, сумма их длин равна N, а средняя их длина равна N/M. Раздельные цепочки удобны на практике потому, что *каждый* список, *скорее всего*, содержит примерно N/M пар ключ-значение. В типичных ситуациях можно проверить это следствие их предположения J и рассчитывать на быстрый поиск и вставку.

Утверждение Л. В хеш-таблице с раздельными цепочками, которая содержит M списков и N ключей, вероятность (в условиях предположения К) того, что количество ключей в списке примерно равно N/M, очень близка к 1.

Набросок доказательства. В силу предположения К, мы имеем применение классической теории вероятностей. Сейчас мы приведем набросок доказательства — для тех читателей, которые знакомы с основами вероятностного анализа. Вероятность того, что данный список будет содержать точно k ключей, описывается *биномиальным распределением* (рис. 3.4.6)

$$\binom{N}{k}\left(\frac{1}{M}\right)^k\left(1-\frac{1}{M}\right)^{N-k}.$$

Это следует из такого рассуждения. Выберем k из N ключей. Эти k ключей хешируются в один конкретный список с вероятностью $1/M$, а другие $N-k$ ключей не хешируются в этот же список с вероятностью $1-1/M$. Обозначив $\alpha = N/M$, можно переписать это выражение в виде

$$\binom{N}{k}\left(\frac{a}{N}\right)^k\left(1-\frac{a}{N}\right)^{N-k}$$

что хорошо аппроксимируется (для небольших α) классическим *распределением Пуассона* (рис. 3.4.7)

$$\frac{\alpha^k e^{-\alpha}}{k!}.$$

Из него следует, что вероятность, что список содержит более $t\alpha$ ключей, ограничена величиной $(\alpha e/t)^t e^{-\alpha}$. Для практических диапазонов параметров эта вероятность крайне мала. Например, если средняя длина списков равно 10, то вероятность того, что в некоторый список хешируются более 20 ключей, меньше $(10e/2)^2 e^{-10} \approx$ $\approx 0{,}0084$. А если средняя длина списков равна 20, то вероятность, что в некоторый список хешируются более 40 ключей, меньше $(20e/2)^2 e^{-20} \approx 0{,}0000016$. Этот результат не гарантирует, что *каждый* список будет коротким. Известно, что при постоянном α средняя длина самого длинного списка растет как $\log N/\log\log N$.

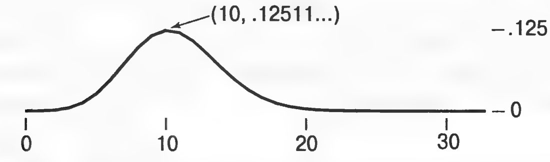


Рис. 3.4.6. *Биномиальное распределение* ($N = 10^4$, $M = 10^3$, $\alpha = 10$)

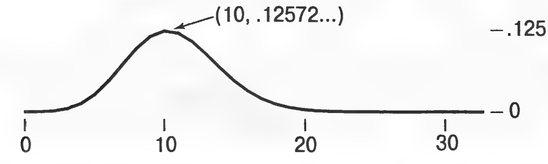


Рис. 3.4.7. *Распределение Пуассона* ($N = 10^4$, $M = 10^3$, $\alpha = 10$)

Этот классический математический результат очень интересен, но важно помнить, что он *полностью зависит* от предположения К. Если хеш-функция не равномерна или не независима, то стоимость поиска и вставки может быть пропорциональна N, т.е. не лучше последовательного поиска. Предположение К гораздо сильнее, чем соответствующие предположения для других уже знакомых нам вероятностных алгоритмов, и его гораздо труднее проверить. Работая с хешированием, мы предполагаем, что каждый ключ, каким сложным он ни был, равновероятно распределяется в один из M индексов. Мы не можем экспериментально проверить каждый возможный ключ, поэтому нужны более изощренные эксперименты со случайными выборками из множества ключей, которые могут появиться в приложении, и последующий статистический анализ. Еще лучше, если в составе теста будет задействован сам алгоритм: это позволит проверить и предположение К, и математические результаты, которые следуют из него.

Утверждение М. В хеш-таблице с раздельными цепочками, которая содержит M списков и N ключей, количество сравнений (проверки на равенство) для промахов и вставок равно ~ N/M.

Обоснование. Для хорошей производительности алгоритма на практике совсем не обязательна функция, полностью равномерная в техническом смысле предположения К. Бесчисленное количество программистов с 1950-х годов наблюдали ускорение, предсказанное утверждением Л, даже для заведомо неравномерных хеш-функций. Например, диаграмма на рис. 3.4.8 показывает, что распределение длин списков для нашего примера `FrequencyCounter` (с реализацией `hash()` на основе `hashCode()` для типа данных `String`) в точности совпадает с теоретической моделью. Хотя об одном исключении упоминалось много раз — это плохая производительность из-за того, что хеш-функция не учитывает все биты ключей. В остальных случаях опыт практических программистов позволяет нам с уверенностью утверждать, что хеширование с раздельными цепочками для массива размером M ускоряет поиск и вставку в таблице имен в M раз.

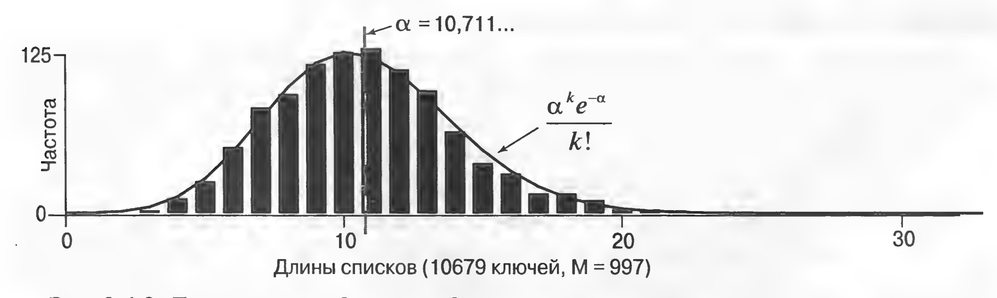


Рис. 3.4.8. *Длины списков для команды* `java FrequencyCounter 8 < tale.txt` *с использованием* `SeparateChainingHashST`

Размер таблицы

В реализации с раздельными цепочками нам нужно выбрать размер таблицы M достаточно малым, чтобы не тратить значительные объемы связной памяти на пустые цепочки, но достаточно большим, чтобы не тратить время на поиск в длинных цепочках. Одним из достоинств раздельных цепочек является то, что это решение не критично: при поступлении большего, чем ожидалось, количества ключей поиски будут занимать немного больше времени, чем для большего размера таблицы; а при вставке лишь не-

большого количества ключей мы получим скоростной поиск за счет дополнительного расхода памяти. Если память не является критическим ресурсом, значение M можно выбрать настолько большим, что время поиска будет константными; а если память ценна, мы все равно имеем ускорение работы в M раз, выбирая M настолько большим, насколько это возможно. Для нашего демонстрационного примера `FrequencyCounter` на рис. 3.4.9 показано (как и ожидалось) сокращение средней стоимости с тысяч сравнений на операцию в случае `SequentialSearchST` до небольшой константы в случае `SeparateChainingHashST`. Еще один вариант — изменение размера массива, чтобы не допустить разрастания списков (см. упражнение 3.4.18).

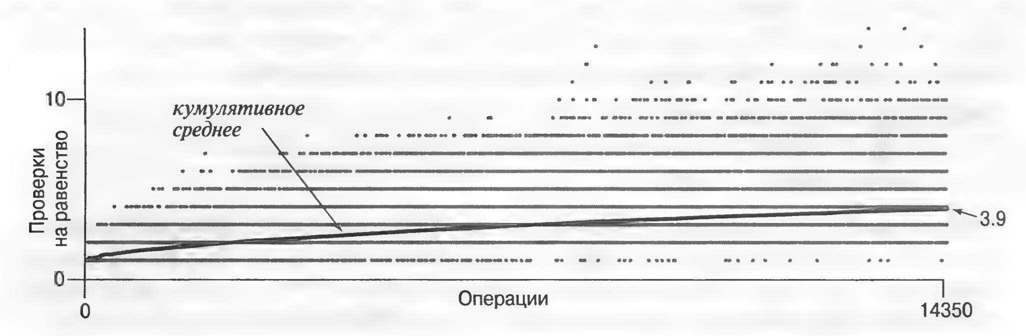


Рис. 3.4.9. Стоимости выполнения команды `java FrequencyCounter 8 < tale.txt` ***с использованием*** `SeparateChainingHashST` ***(M = 997)***

Удаление

Чтобы удалить пару ключ-значение, нужно просто найти объект `SequentialSearchST`, содержащий ключ, а затем вызвать метод `delete()` для этой таблицы (см. упражнение 3.1.5). Подобное повторное использование кода позволяет легко реализовать эту базовую операцию для связных списков.

Операции, основанные на упорядоченности

Хеширование основано на равномерном распределении ключей, поэтому любая упорядоченность ключей при хешировании теряется. Если нужно быстро найти наибольший или наименьший ключ, найти ключи в заданном диапазоне или реализовать любую другую операцию из API упорядоченной таблицы имен (рис. 3.1.2), то хеширование для этого *не годится*, т.к. эти операции потребуют линейного времени.

Хеширование с раздельными цепочками легко реализовать, и, возможно, это самая быстрая (и широко применяемая) реализация таблиц имен для приложений, где порядок ключей не важен. Если ключи являются встроенными типами Java или вашим собственным типом с хорошо проверенной реализацией `hashCode()`, то алгоритм 3.5 предоставляет быстрый и легкий способ для быстрого поиска и вставки. А теперь мы рассмотрим альтернативную схему для разрешения коллизий, которая тоже эффективна.

Хеширование с линейным опробованием

Другой способ реализации хеширования — хранение N пар ключ-значение в хеш-таблице размером $M > N$, используя для разрешения коллизий пустые элементы таблицы. Такие методы называются методами хеширования *с открытой адресацией*.

Простейший из методов с открытой адресацией называется *линейным опробованием*: при возникновении коллизии (когда хеширование дает индекс таблицы, уже занятый ключом, отличным от искомого) мы просто проверяем следующий элемент таблицы (увеличив индекс). Линейное опробование характеризуется тремя возможными результатами проверки элемента:

- ключ равен искомому — попадание;
- пустая позиция (нулевой ключ в индексируемой позиции) — промах;
- ключ не равен искомому — проверяем следующий элемент.

То есть мы хешируем ключ в индекс таблицы, проверяем, равен ли находящийся там ключ искомому, и продолжаем проверки (наращивая индекс и возвращаясь в начало таблицы при достижении ее конца), пока не обнаружим искомый ключ или пустой элемент таблицы. Операцию, которая определяет, содержит ли данный элемент таблицы ключ, равный искомому, обычно называют *пробой*. Этот термин мы будем использовать наряду с уже привычным термином *сравнение*, хотя некоторые пробы выполняют проверки на `null`.

Принцип, на котором основано хеширование с открытой адресацией, таков: вместо затрат памяти на ссылки в связных списках, память тратится на пустые элементы в хеш-таблице, которые означают завершение последовательностей проб. Как видно из кода `LinearProbingHashST` (алгоритм 3.6; рис. 3.4.10), реализация API таблицы имен на основе этой идеи совсем не сложна.

Ключ	Хеш	Значение	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S	6	0							**S** **0**									
E	10	1							S 0				**E** **1**					
A	4	2					**A** **2**		S 0				E 1					
R	14	3					A 2		S 0				E 1				**R** **3**	
C	5	4					A 2	**C** **5**	S 0				E 1				R 3	
H	4	5					A 2	C 5	S 0	**H** **5**			E 1				R 3	
E	10	6					A 2	C 5	S 0	H 5			E (6)				R 3	
X	15	7					A 2	C 5	S 0	H 5			E 6				R 3	**X** **7**
A	4	8					A (8)	C 5	S 0	H 5			E 6				R 3	X 7
M	1	9		**M** **9**			A 8	C 5	S 0	H 5			E 6				R 3	X 7
P	14	10	**P** **10**	M 9			A 8	C 5	S 0	H 5			E 6				R 3	X 7
L	6	11	P 10	M 9			A 8	C 5	S 0	H 5	**L** **11**		E 6				R 3	X 7
E	10	12	P 10	M 9			A 8	C 5	S 0	H 5	L 11		E (12)				R 3	X 7

Полужирные элементы — новые

серые элементы не участвуют

Обычные элементы — пробы

последовательность проб продолжается с 0

keys[]

vals[]

Рис. 3.4.10. *Трассировка реализации таблицы имен с линейным опробованием для стандартного клиента индексации*

В этой реализации задействованы два параллельных массива — один для ключей, а другой для значений — а хеш-функция используется в качестве индекса для обращения к данным.

Листинг 3.4.4. Алгоритм 3.6. Хеширование с линейным опробованием

```
public class  LinearProbingHashST<Key, Value>
{
   private int N;                 // количество пар ключ-значение в таблице
   private int M = 16;            // размер таблицы с линейным опробованием
   private Key[] keys;            // ключи
   private Value[] vals;          // значения

   public LinearProbingHashST()
   {
      keys = (Key[])   new Object[M];
      vals = (Value[]) new Object[M];
   }

   private int hash(Key key)
   {  return (key.hashCode() & 0x7fffffff) % M; }

   private void resize()          // См. листинг 3.4.6.

   public void put(Key key, Value val)
   {
      if (N >= M/2) resize(2*M); // удвоение M (см. текст)
      int i;
      for (i = hash(key); keys[i] != null; i = (i + 1) % M)
         if (keys[i].equals(key)) { vals[i] = val; return; }
      keys[i] = key;
      vals[i] = val;
      N++;
   }

   public Value get(Key key)
   {
      for (int i = hash(key); keys[i] != null; i = (i + 1) % M)
         if (keys[i].equals(key))
            return vals[i];
      return null;
   }
}
```

В этой реализации таблицы имен ключи и значения хранятся в параллельных массивах (как в классе `BinarySearchST`), но в качестве признака окончания кластеров ключей используются пустые места (помеченные ключом `null`). Если новый ключ хешируется в пустой элемент, он сохраняется в нем, иначе выполняется последовательный поиск пустой позиции. При поиске ключа выполняется последовательный просмотр, начиная с хеш-индекса, пока не встретится `null` (промах) или искомый ключ (попадание). Реализация метода `keys()` вынесена в упражнение 3.4.19.

Удаление

Как можно удалить пару ключ-значение из таблицы с линейным опробованием? Немного поразмыслив, становится понятно, что простое занесение в позицию ключа значения `null` не годится, т.к. это может преждевременно прекратить поиск для ключа, который был вставлен позже. В качестве примера представьте, что в нашем примере трассировки (рис. 3.4.10) удален таким способом ключ C, а затем выполняется поиск ключа H. Хеш-значение для H равно 4, но этот ключ находится в конце кластера, в позиции 7. Если в позицию 5 занести `null`, то метод `get()` не найдет ключ H. Тогда придется заново вставить в таблицу все ключи данного кластера справа от удаленного ключа. Этот процесс не так прост, как кажется, поэтому мы рекомендуем проработать код из листинга 3.4.5 (см. также упражнение 3.4.17).

Листинг 3.4.5. Удаление для линейного опробования

```
public void delete(Key key)
{
  if (!contains(key)) return;
  int i = hash(key);
  while (!key.equals(keys[i]))
    i = (i + 1) % M;
  keys[i] = null;
  vals[i] = null;
  i = (i + 1) % M;
  while (keys[i] != null)
  {
    Key   keyToRedo = keys[i];
    Value valToRedo = vals[i];
    keys[i] = null;
    vals[i] = null;
    N--;
    put(keyToRedo, valToRedo);
    i = (i + 1) % M;
  }
  N--;
  if (N > 0 N == M/8) resize(M/2);
}
```

Как и в случае раздельных цепочек, производительность хеширования с открытой адресацией зависит от отношения $\alpha = N/M$, но интерпретируется оно по-другому. Теперь α называется *коэффициентом загрузки* хеш-таблицы. Для раздельных цепочек α — это среднее количество ключей в списке, и обычно оно больше 1. А в линейном опробовании α представляет собой долю занятых элементов таблицы, и он не может быть больше 1. Вообще говоря, в коде `LinearProbingHashST` нельзя допускать, чтобы коэффициент загрузки дорос до 1 (полностью занятая таблица), т.к. промах при поиске уйдет в бесконечный цикл по полной таблице. Поэтому по соображениям производительности необходимо использовать изменение размера массива, чтобы гарантировать, что коэффициент загрузки был от одной восьмой до одной второй. Эта стратегия подкреплена математическим анализом, который мы рассмотрим, прежде чем перейти к рассмотрению деталей реализации.

Кластеризация

Средняя стоимость линейного опробования зависит от характера группировки элементов при вставке в непрерывные группы занятых элементов таблицы — они называются *кластерами*. Например, при вставке ключа C в нашем примере получится кластер (A C S) длиной 3. И получается, что для вставки ключа H понадобятся четыре пробы, т.к. H хешируется в первую позицию кластера. Понятно, что для хорошей производительности нужны короткие кластеры. Это требование становится проблематичным, когда таблица заполняется, т.к. в ней становится все больше длинных кластеров (рис. 3.4.11 и 3.4.12). Более того, поскольку все позиции таблицы с равной вероятностью могут быть хеш-значением следующего вставляемого ключа (если выполняется предположение о равномерном хешировании), то *более вероятно* удлинение длинных, а не коротких кластеров, т.к. хеширование нового ключа в любой элемент кластера увеличит его длину на 1 (а, возможно, и значительно больше, если этот кластер отделялся от другого лишь одной пустой позицией). Теперь мы попробуем численно выразить эффект кластеризации для предсказания производительности линейного опробования и использовать это знание для задания параметров при проектировании наших реализаций.

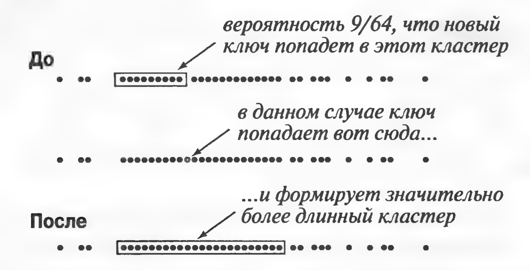


Рис. 3.4.11. Кластеризация при линейном опробовании (M = 64)

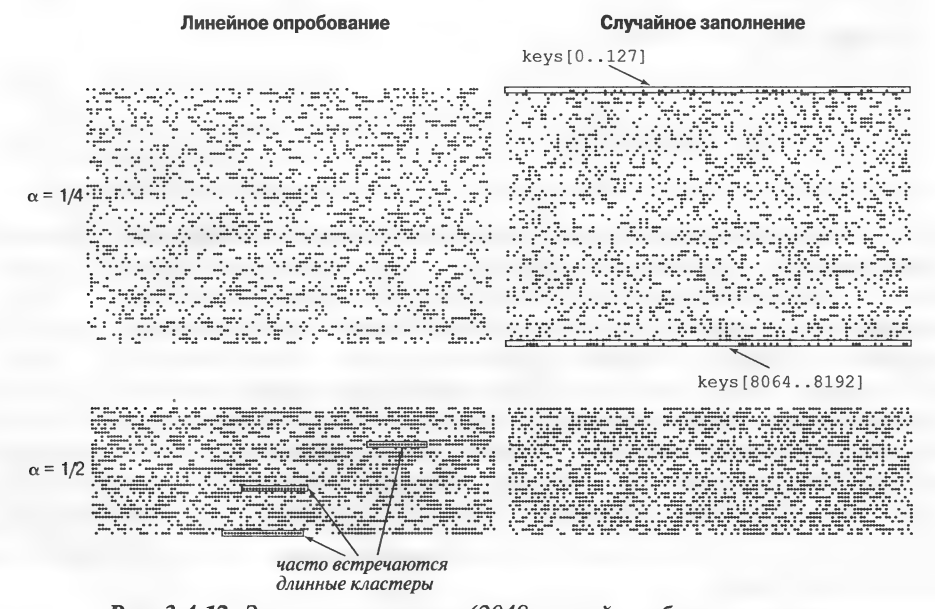


Рис. 3.4.12. Занятость памяти (2048 ключей, таблицы свернуты в 128-элементные строки)

Анализ линейного опробования

Несмотря на относительно простой вид результатов, точный анализ линейного опробования представляет собой очень сложную задачу. Вывод Кнутом приведенных ниже формул в 1962 г. был значительной вехой в анализе алгоритмов.

Утверждение Н. В хеш-таблице с линейным опробованием с M элементами и $N = \alpha M$ ключами среднее количество проб (если выполняется предположение J) равно

$$\sim \frac{1}{2}\left(1+\frac{1}{1-a}\right) \text{ и } \sim \frac{1}{2}\left(1+\frac{1}{(1-a)^2}\right)$$

для попаданий и промахов (или вставок) соответственно. В частности, когда α примерно равно 1/2, среднее количество проб для попадания равно примерно 3/2, а для промаха — примерно 5/2. Эти оценки не так точны при приближении α к 1, но в таком случае они нам и не нужны, т.к. мы собираемся выполнять линейное опробование только для значений α, меньших одной второй.

Обсуждение. Для вычисления среднего значения мы вычисляем стоимость промахов, начиная в каждой позиции таблицы, а затем делим общий итог на M. Для всех промахов необходима хотя бы одна проба, поэтому мы будем подсчитывать количество проб после первой. Рассмотрим два следующих крайних случая в таблице с линейным опробованием, которая заполнена наполовину ($M = 2N$). В лучшем случае позиции таблицы с четными индексами будут пусты, а с нечетными — заняты. В худшем случае позиции в первой половине таблицы будут пусты, а во второй половине заняты. Средняя длина кластеров в обоих случаях равна $N/(2N) = 1/2$, но среднее количество проб для промаха равно 1 (все поиски выполняют хотя бы одну пробу) плюс $(0 + 1 + 0 + 1 + ...)/(2N) = 1/2$ в лучшем случае, и плюс $(N + (N-1) + ...) / (2N) \sim N/4$ в худшем случае. Это рассуждение можно обобщить и показать, что среднее количество проб для промаха пропорционально *квадратам* длин кластеров: если длина кластера равна t, то вклад данного кластера в общий итог задается выражением $(t + (t-1) + ... + 2 + 1) / M = t(t + 1) / (2M)$. Сумма длин кластеров равна N, поэтому, добавив эту стоимость ко всем элементам таблицы, мы найдем, что общая средняя стоимость для промаха равна $1 + N/(2M)$ плюс сумма квадратов длин кластеров, деленная на $2M$. Значит, для любой заданной таблицы мы можем быстро вычислить среднюю стоимость промаха в этой таблице (см. упражнение 3.4.21). В общем случае кластеры формируются сложным динамическим процессом (алгоритм линейного опробования), который трудно охарактеризовать аналитически, и этот анализ выходит за рамки данной книги.

Утверждение Н гласит, что (как обычно, в условиях предположении K) в почти заполненной таблице можно ожидать, что поиск потребует огромного количества проб: по мере приближения α к 1 значения в формулах, описывающих количество проб, быстро растут. Однако если обеспечить, что коэффициент загрузки не будет превышать 1/2, то ожидаемое количество проб будет от 1,5 до 2,5. Теперь мы рассмотрим изменение размера массива для этой цели.

Изменение размера массива

Чтобы коэффициент загрузки никогда не превышал одну вторую, можно использовать наш стандартный прием изменения размера массива из главы 1. Вначале нам понадобится новый конструктор для класса `LinearProbingHashST`, который принимает в качестве аргумента фиксированную емкость таблицы: в конструктор в алгоритме 3.6 нужно добавить строку, которая присваивает переменной `M` заданное значение перед созданием массивов. Еще нам понадобится метод `resize()` из листинга 3.4.6, который создает новый объект `LinearProbingHashST` указанного размера, копирует все ключи и значения из старой таблицы в новую, а затем заново хеширует все ключи в новой таблице. Эти добавления позволяют реализовать удвоение размера массива. Вызов `resize()` в первом операторе метода `put()` гарантирует, что таблица заполнена менее чем наполовину. Этот код строит хеш-таблицу удвоенного размера с теми же ключами, таким образом уменьшая вдвое значение α. Как и в других случаях изменения размера массива, в конец метода `delete()` необходимо добавить оператор

```
if (N > 0 && N <= M/8) resize(M/2);
```

чтобы таблица была заполнена, по крайней мере, на одну восьмую. Это гарантирует, что объем задействованной памяти будет всегда отличаться от количества пар ключ-значение в таблице не более чем на постоянный множитель. При использовании массивов переменных размеров мы уверены, что $\alpha \leq 1/2$.

Раздельные цепочки

Этот же метод годится и для ограничения длины списков (средняя длина от 2 до 8) и в случае раздельных цепочек. Нужно в методе `resize()` заменить `LinearProbingHashST` на `SeparateChainingHashST`, в методе `put()` вызывать `resize(2*M)`, если `N >= M/2`, а в методе `delete()` вызывать `resize(M/2)`, если `N > 0 && N <= M/8`. В случае раздельных цепочек изменение размера массива *не обязательно*, и его не стоит выполнять, если имеется приличная оценка от клиента N: нужно просто выбрать размер таблицы равным M, помня, что время поиска пропорционально $1 + N/M$. Для линейного опробования изменение размера массива *необходимо*. Если клиент вставит больше пар ключ-значение, чем вы ожидали, может получить не только слишком большое время поиска, но и бесконечный цикл при полном заполнении.

Амортизационный анализ

С теоретической точки зрения при применении изменения размера массива необходимо рассматривать амортизированную границу, ведь вставки, которые приводят к удвоению таблицы, требуют большего количества проб.

Утверждение О. Пусть хеш-таблица построена с помощью изменения размера массива, начиная с пустой таблицы. В условиях предположения К любая последовательность t операций *поиска*, *вставок* и *удалений* в таблице имен выполняется за ожидаемое время, пропорциональное t, и с затратами памяти, всегда пропорциональными количеству ключей в таблице с постоянным коэффициентом.

Доказательство. И для раздельных цепочек, и для линейного опробования этот факт следует из простого повторения амортизационного анализа для роста массива, который впервые был проведен в главе 1, вместе с утверждением Л и утверждением Н.

Графики кумулятивных средних для клиента FrequencyCounter (рис. 3.4.13 и 3.4.14) хорошо иллюстрируют динамическое поведение изменения размера массива при хешировании. При каждом удвоении размера массива кумулятивное среднее увеличивается примерно на 1, т.к. выполняется новое хеширование всех ключей в таблице, а затем оно снижается, т.к. в каждую позицию таблицы хешируется лишь половина ключей, но такое снижение замедляется по мере заполнения таблицы.



Рис. 3.4.13. Стоимости выполнения команды** `java FrequencyCounter 8 < tale.txt` **с использованием** `SeparateChainingHashST` **(с удвоением размера)



Рис. 3.4.14. Стоимости выполнения команды** `java FrequencyCounter 8 < tale.txt` **с использованием** `LinearProbingHashST` **(с удвоением размера)

Память

Как уже было сказано, понимание требования к памяти является важным фактором, если необходимо настроить хеширующие алгоритмы на оптимальную производительность. Такая настройка — удел экспертов, но можно получить грубую оценку требуемого объема памяти, вычислив количество обращений. Если не учитывать память для ключей и значений, то наша реализация SeparateChainingHashST использует память для M ссылок на объекты SequentialSearchST плюс память для самих M объектов SequentialSearchST. Каждый объект SequentialSearchST обычно содержит 16 байтов служебной информации в объекте плюс одна 8-байтовая ссылка (first), и еще N объектов Node, каждый из которых содержит 24 служебных байта плюс 3 ссылки (key, value и next). Это сравнимо с дополнительной ссылкой на узел в деревьях бинарного

поиска. Если размер массива поддерживает заполнение от 1/8 до 1/2, то линейное опробование использует от $4N$ до $16N$ ссылок. Так что с точки зрения использования памяти выбор хеширования обычно не оправдан (табл. 3.4.1). Эти расчеты несколько отличаются для примитивных типов (см. упражнение 3.4.24).

Таблица 3.4.1. Использование памяти в таблицах имен

Метод	Объем памяти для N элементов (ссылочные типы)
Раздельные цепочки	$\sim 48N + 64M$
Линейное опробование	от $\sim 32N$ до $\sim 128N$
ДБП	$\sim 56N$

Начиная с первых дней компьютерных вычислений, исследователи изучали (и изучают) хеширование и нашли много способов улучшить рассмотренные нами базовые алгоритмы. По данной теме имеется обширная литература. В основном эти усовершенствования снижают требования к памяти или времени: можно получить то же время поиска, но с меньшими затратами памяти, или ускорить поиск с тем же объемом памяти. Есть усовершенствования, которые обеспечивают лучшие гарантии для стоимости поиска в худшем случае. Некоторые улучшают построение хеш-функций. Часть этих методов вынесена в упражнения.

Точное сравнение раздельных цепочек и линейного опробования зависит от множества деталей реализации и от требования клиента к памяти и времени. Обычно раздельные цепочки проигрывают линейному опробованию с точки зрения производительности (см. упражнение 3.5.31). На практике основное различие между этими двумя методами состоит в том, что раздельные цепочки используют небольшой блок памяти для каждой пары ключ-значение, а линейное опробование использует два больших массива для всей таблицы. Для больших таблиц эти потребности означают совершенно разные требования к системе управления памятью. В современных системах такие компромиссы изучаются экспертами по ситуациям с сильной зависимостью от производительности.

При использовании хеширования в разумных условиях вполне резонно ожидать выполнения операций поиска и вставки в таблице имен за константное время, независимо от размера таблицы. Это ожидание — теоретически оптимальная производительность для любой реализации таблицы имен. Но все же хеширование не является панацеей по нескольким причинам.

- Нужна хорошая хеш-функция для каждого типа ключа.
- Гарантия производительности зависит от качества хеш-функции.
- Некоторые хеш-функции трудно и долго вычислять.
- Практически нет поддержки операций таблиц имен, основанных на упорядоченности.

Сравнение хеширования с другими способами организации таблиц имен, выходящее за эти простые соображения, будет выполнено в начале раздела 3.5.

Вопросы и ответы

Вопрос. **Как реализован в Java метод `hashCode()` для типов `Integer`, `Double` и `Long`?**

Ответ. Для `Integer` просто возвращается 32-битное значение. Для типов `Double` и `Long` возвращается результат *исключающего или* первых и последних 32 битов стандартного машинного представления числа. Эти способы не очень-то случайны, но они хорошо справляются с распределением значений.

Вопрос. **При изменении размера массива размер таблицы всегда равен степени 2. Но ведь тогда используются лишь наименее значимые биты `hashCode()`?**

Ответ. Да, эта проблема существует, особенно в стандартных реализациях. Один из способов справиться с ней состоит в предварительном распределении значений ключей с помощью простого числа, большего `M`:

```
private int hash(Key x)
{
   int t = x.hashCode() & 0x7fffffff;
   if (lgM < 26) t = t % primes[lgM+5];
   return t % M;
}
```

Здесь используется переменная экземпляров `lgM`, равная $\lg M$ (инициализируется подходящим значением, а затем увеличивается при удвоении размера и уменьшается при сжатии таблицы), и массив `primes[]`, который содержит наименьшие простые числа, бóльшие каждой степени 2 (см. табл. 3.4.2). Неплохим выбором будет константа 5: первая операция `%` равномерно распределяет значения по всем числам, меньшим простого, а вторая распределяет примерно пять этих значений на каждое значение, меньшее `M`. Для больших `M` этот прием не всегда удачен.

Таблица 3.4.2. Простые числа для размеров хеш-таблиц

k	δ_k	`primes[k]` $(2^k - \delta_k)$
5	1	31
6	3	61
7	1	127
8	5	251
9	3	509
10	3	1021
11	9	2039
12	3	4093
13	1	8191
14	3	16381
15	19	32749
16	15	65521
17	1	131071
18	5	262139

Окончание табл. 3.4.2

k	δ_k	`primes[k]` $(2^k - \delta_k)$
19	1	524287
20	3	1048573
21	9	2097143
22	3	4194301
23	15	8388593
24	3	16777213
25	39	33554393
26	5	67108859
27	39	134217689
28	57	268435399
29	3	536870909
30	35	1073741789
31	1	2147483647

Вопрос. **Почему мы не реализуем `hash(x)` с помощью операции `x.hashCode()%M`?**

Ответ. Нужен результат от `0` до `M-1`, но в Java функция `%` может давать отрицательный результат.

Вопрос. **Тогда почему не реализовать `hash(x)`, возвращая `Math.abs(x.hashcode())%M`?**

Ответ. Хорошая попытка. К сожалению, функция `Math.abs()` возвращает отрицательный результат для наибольшего отрицательного числа. Для большинства типичных приложений это переполнение не является проблемой, но при хешировании ваша программа может завершиться аварийно после миллиарда вставок, что вряд ли кому-то понравится. Например, значение `s.hashCode()` равно -2^{31} для Java-строки `"polygenelubricants"`. Поиск других строк, которые хешируются в это значение (и в 0) — интересное развлечение в области алгоритмов.

Вопрос. **Почему в алгоритме 3.5 не использовать `BinarySearchST` или `RedBlackBST` вместо `SequentialSearchST`?**

Ответ. Обычно параметры хеш-таблицы выбираются так, чтобы количество ключей, хешируемых в каждое значение, было небольшим, и в таких случаях обычно лучше использовать элементарные таблицы имен. В некоторых ситуациях подобные гибридные методы могут несколько повысить производительность, но такую тонкую настройку лучше оставить экспертам.

Вопрос. **Быстрее ли хеширование, чем поиск в красно-черных ДБП?**

Ответ. Это зависит от типа ключа, который определяет стоимость вычисления `hashCode()` в сравнении со стоимостью `compareTo()`. Для часто встречающихся типов ключей и для стандартных реализаций Java эти стоимости сопоставимы, поэтому хеширование работает значительно быстрее, т.к. в нем выполняется лишь постоянное количество операций. Однако все не так, если нужны операции, основанные на упорядоченности, для которых нет эффективной поддержки в хеш-таблицах. Подробнее см. в разделе 3.5.

Вопрос. **Почему не позволить заполнение таблицы при линейном опробовании, скажем, на три четверти?**

Ответ. Нет особых причин. Вы вольны выбрать любое значение α и воспользоваться утверждением Н для оценки затрат на поиск. Для $\alpha = 3/4$ средняя стоимость попадания равна 2,5, а промаха — 8,5, но при $\alpha = 7/8$ средняя стоимость промаха возрастает до 32,5 — вряд ли это вам понравится. При приближении α к 1 оценка в утверждении Н перестает быть верной, но никто и не допустит такой степени заполнения таблицы.

Упражнения

3.4.1. Вставьте ключи `E A S Y Q U T I O N` в указанном порядке в первоначально пустую таблицу с $M = 5$ списками раздельных цепочек. Для преобразования k-й буквы алфавита в индекс таблицы используйте хеш-функцию `11 k % M`.

3.4.2. Разработайте альтернативную реализацию `SeparateChainingHashST`, в которой напрямую используется код работы со связными списками из класса `SequentialSearchST`.

3.4.3. Добавьте в реализацию из предыдущего упражнения целочисленное поле для каждой пары ключ-значение, которое содержит количество элементов в таблице на момент вставки этой пары. Затем реализуйте метод удаления всех ключей (и связанных с ними значений), у которых это поле содержит значение, большее заданного целого `k`. *Примечание*: эта дополнительная возможность удобна при реализации таблиц имен для компиляторов.

3.4.4. Напишите программу для определения значений `a` и `M` с наименьшим `M`, чтобы хеш-функция `(a * k) % M` преобразования k-й буквы алфавита в индекс таблицы давала различные значения (без коллизий) для ключей `S E A R C H X M P L`. Такой результат называется *идеальной хеш-функцией*.

3.4.5. Допустима ли такая реализация функции `hashCode()`?

```
public int hashCode()
{  return 17;  }
```

Если да, опишите эффект ее использования. Если нет, объясните, почему.

3.4.6. Пусть ключи являются t-битными целыми числами. Для модульной хеш-функции с простым значением M докажите для каждого бита ключа, что если существуют два ключа, различающимся только этим битом, то они имеют различные хеш-значения.

3.4.7. Проанализируйте идею реализации модульного хеширования с целочисленными ключами с помощью кода `(a * k) % M`, где `a` — произвольное фиксированное простое число. Достаточно ли эффекта такого перемешивания для использования не простого значения `M`?

3.4.8. Сколько пустых списков можно ожидать после вставки N ключей в хеш-таблицу типа `SeparateChainingHashST` для $N = 10, 10^2, 10^3, 10^4, 10^5$ и 10^6? ***Совет***: см. упражнение 2.5.31.

3.4.9. Реализуйте энергичный метод `delete()` для класса `SeparateChainingHashST`.

3.4.10. Вставьте ключи `E A S Y Q U T I O N` в указанном порядке в первоначально пустую таблицу размером $M = 5$ с линейным опробованием. Для преобразования k-й буквы алфавита в индекс таблицы используйте хеш-функцию `11 k % M`. Выполните упражнение еще раз для $M = 10$.

3.4.11. Приведите содержимое хеш-таблицы с линейным опробованием, которая получится после вставки ключей `E A S Y Q U T I O N` в указанном порядке в первоначально пустую таблицу с первоначальным размером $M = 4$, которая удваивает размер при заполнении наполовину. Для преобразования k-й буквы алфавита в индекс таблицы используйте хеш-функцию `11 k % M`.

3.4.12. Пусть ключи от `A` до `G` с хеш-значениями, приведенными ниже, вставляются в некотором порядке в первоначально пустую таблицу размером 7 с линейным опробованием (без изменения размера). Какой из приведенных ниже вариантов не может получиться в результате вставки этих ключей?

а) `E F G A C B D`

б) `C E B G F D A`

в) `B D F A C E G`

г) `C G B A D E F`

д) `F G B D A C E`

е) `G E C A D B F`

Приведите минимальное и максимальное количество проб, которое может понадобиться для построения таблицы размером 7 для этих ключей, и порядок вставки, который обосновывает ваш ответ.

3.4.13. Какая из приведенных ниже ситуаций приведет к ожидаемому *линейному* времени выполнения для случайных попаданий в хеш-таблице с линейным опробованием?

а) Все ключи хешируются в один и тот же индекс.

б) Все ключи хешируются в различные индексы.

в) Все ключи хешируются в четные индексы.

г) Все ключи хешируются в различные четные индексы.

3.4.14. Ответьте на вопрос предыдущего упражнения для *промахов*, считая, что искомый ключ равновероятно хешируется в каждую позицию таблицы.

3.4.15. Сколько сравнений понадобится в худшем случае для вставки N ключей в первоначально пустую таблицу с линейным опробованием и изменением размера?

3.4.16. Пусть таблица с линейным опробованием размера 10^6 наполовину заполнена, причем занятые позиции распределены случайным образом. Оцените вероятность того, что заняты все позиции с индексами, кратными 100.

3.4.17. Приведите результат использования метода `delete()` из листинга 3.4.5 для удаления ключа `C` из таблицы типа `LinearProbingHashST` с нашим стандартным клиентом индексации (см. рис. 3.4.10).

3.4.18. Добавьте в класс SeparateChainingHashST конструктор, который позволит клиенту задать среднее количество проб, еще приемлемое для поисков. Используйте изменение размера массива, чтобы поддерживать средний размер списка меньше указанного значения, и используйте прием, описанный в разделе "Вопросы и ответы" и табл. 3.4.2, для выбора простого значения модуля для функции hash().

3.4.19. Реализуйте метод keys() для классов SeparateChainingHashST и LinearProbingHashST.

3.4.20. Добавьте в класс LinearProbingHashST метод, который вычисляет среднюю стоимость попадания в таблице, считая, что поиск может быть выполнен для каждого ключа в таблице с равной вероятностью.

3.4.21. Добавьте в класс LinearProbingHashST метод, который вычисляет среднюю стоимость *промаха* в таблице, считая, что используется случайная хеш-функция. *Примечание*: для выполнения этого упражнения не обязательно вычислять какую-либо хеш-функцию.

3.4.22. Реализуйте функцию hashCode() для типов Point2D, Interval, Interval2D и Date.

3.4.23. Пусть используется модульное хеширование для строковых ключей с R = 256 и M = 255. Покажите, что этот выбор неудачен, т.к. любая перестановка букв в строке хешируется в одно и то же значение.

3.4.24. Проанализируйте затраты памяти при использовании раздельных цепочек, линейного опробования и ДБП для ключей double. Оформите результаты в виде таблицы, подобной табл. 3.4.1.

Творческие задачи

3.4.25. *Кеш хешей*. Добавьте в код Transaction (листинг 3.4.2) переменную экземпляра hash, в которой метод hashCode() может сохранить хеш-значение при первом вычислении для каждого объекта, чтобы при последующих вызовах не тратить время на повторное вычисление. *Внимание*: эта идея пригодна только для неизменяемых типов.

3.4.26. *Ленивое удаление для линейного опробования*. Добавьте в класс LinearProbingHashST метод delete(), удаляющий пару ключ-значение с помощью занесения значения null (но без удаления ключа). Такая пара удаляется из таблицы при изменении ее размера методом resize(). Главная проблема — решить, когда вызвать resize(). *Примечание*: значение null следует перезаписать, если последующая операция put() связывает с этим ключом новое значение. Ваша программа должна подсчитывать число таких *перезаписанных* элементов, а также пустых позиций, при принятии решения об увеличении или уменьшении размера таблицы.

3.4.27. *Двойное опробование*. Добавьте в класс SeparateChainingHashST вторую хеш-функцию с выбором более короткого из двух списков. Приведите трассировку процесса вставки ключей E A S Y Q U T I O N в указанном порядке в первоначально пустую таблицу размером $M = 3$ с преобразованиями k-го символа 11 k % M в первой хеш-функции и 17 k % M во второй. Приведите среднее количество проб для случайных попаданий и промахов в такой таблице.

3.4.28. *Двойное хеширование.* Добавьте в класс `LinearProbingHashST` использование второй хеш-функции, которая задает последовательность проб. А именно, замените оба выражения `(i + 1) % M` на `(i + k) % M`, где `k` — ненулевое число, зависящее от ключа и взаимно простое с `M`. *Примечание*: последнее условие удовлетворяется автоматически, если `M` простое. Приведите трассировку процесса вставки ключей `E A S Y Q U T I O N` в указанном порядке в первоначально пустую таблицу размером $M = 11$ и с хеш-функциями из предыдущего упражнения. Приведите среднее количество проб для случайных попаданий и промахов в такой таблице.

3.4.29. *Удаление.* Реализуйте энергичный метод `delete()` для таблиц из двух предыдущих упражнений.

3.4.30. *Критерий хи-квадрат.* Добавьте в класс `SeparateChainingHashST` метод, вычисляющий критерий χ^2 (Пирсона) для хеш-таблицы. При наличии N ключей в таблице размером M это число определяется формулой

$$\chi^2 = (M/N)\ [\ (f_0 - N/M)^2 + (f_1 - N/M)^2 + \ldots + (f_{M-1} - N/M)^2\ ]$$

где f_i — количество ключей с хеш-значением i. Такой критерий позволяет проверить предположение, что хеш-функция генерирует случайные значения. Если они действительно случайны, то при $N > cM$ значение χ^2 должно находиться в пределах от $M-\sqrt{M}$ и $M+\sqrt{M}$ с вероятностью $1-1/c$.

3.4.31. *Кукушкино хеширование.* Разработайте реализацию таблицы имен, в которой используются две хеш-таблицы и две хеш-функции. Любой заданный ключ может находиться в одной из таблиц, но не в обеих. При вставке нового ключа хешируйте его в одну из таблиц. Если позиция в этой таблице занята, замените ключ в ней новым ключом, а старый ключ хешируйте в другую таблицу (где опять возможно вышвыривание ключа, который находится там). Если процесс зациклился, начните его снова. Поддерживайте заполнение таблиц менее чем наполовину. Этот метод использует константное количество проверок на равенство в худшем случае для поиска (в обычном смысле) и амортизированно константное время для вставок.

3.4.32. *Хеш-атака.* Найдите 2^N строк, каждая длиной 2^N, которые имеют одинаковое значение `hashCode()` для следующей реализации:

```
public int hashCode()
{
   int hash = 0;
   for (int i = 0; i < length(); i ++)
      hash = (hash * 31) + charAt(i);
   return hash;
}
```

Подсказка: строки `Aa` и `BB` дают одинаковые значения.

3.4.33. *Неудачная хеш-функция.* Вот реализация `hashCode()` для типа `String`, которая применялась в ранних версиях Java:

```
public int hashCode()
{
   int hash = 0;
   int skip = Math.max(1, length()/8);
   for (int i = 0; i < length(); i += skip)
      hash = (hash * 37) + charAt(i);
   return hash;
}
```

Как вы думаете, почему проектировщики выбрали такую реализацию и почему впоследствии от нее отказались в пользу реализации из предыдущего упражнения?

Эксперименты

3.4.34. *Стоимость хеширования.* Экспериментально определите отношение времени, которое необходимо функции `hash()`, ко времени, которое необходимо функции `compareTo()`, для как можно большего количества обычно применяемых типов ключей, для которых можно получить осмысленные результаты.

3.4.35. *Критерий хи-квадрат.* Используйте решение из упражнения 3.4.30 для проверки предположения, что хеш-функции для распространенных типов ключей генерируют случайные значения.

3.4.36. *Диапазон длин списков.* Напишите программу, которая вставляет N ключей `int` в таблицу размером $N/100$ с раздельными цепочками, а затем определяет длину самого короткого и самого длинного списков, для $N = 10^3$, 10^4, 10^5 и 10^6.

3.4.37. *Гибрид.* Экспериментально оцените эффект использования объектов `RedBlackBST` вместо `SequentialSearchST` для разрешения коллизий в классе `Separate ChainingHashST`. Это решение гарантированно имеет логарифмическую производительность даже для крайне неудачной хеш-функции, но в нем придется использовать две различных реализации таблицы имен. Каков практический эффект?

3.4.38. *Распределение с раздельными цепочками.* Напишите программу, которая вставляет 10^5 случайных неотрицательных целых чисел, меньших 10^6, в таблицу размером 10^5 с раздельными цепочками и выводит график общего количества проб при каждых 10^3 последовательных вставках. Насколько полученные результаты подтверждают утверждение Л?

3.4.39. *Распределение с линейным опробованием.* Напишите программу, которая вставляет $N/2$ случайных ключей типа `int` в таблицу размером N с линейным опробованием, а затем вычисляет среднюю стоимость промахов в полученной таблице в зависимости от длин кластеров, для $N = 10^3$, 10^4, 10^5 и 10^6.

3.4.40. *Графики.* Добавьте в классы `LinearProbingHashST` и `SeparateChainingHashST` возможность вывода графиков наподобие приведенных в тексте.

3.4.41. *Двойное опробование.* Экспериментально оцените эффективность двойного опробования (см. упражнение 3.4.27).

3.4.42. *Двойное хеширование.* Экспериментально оцените эффективность двойного хеширования (см. упражнение 3.4.28).

3.4.43. *Задача парковки* (Д. Кнут). Экспериментально подтвердите гипотезу, что количество сравнений, необходимых для вставки M случайных ключей в таблицу размером M с линейным опробованием равно $\sim cM^{3/2}$, где $c = \sqrt{\pi/2}$.

3.5. Применения

Еще на заре компьютерных вычислений, когда таблицы имен позволили программистам перейти от использования числовых адресов в машинном языке к символическим именам в языке ассемблера, и до современных приложений в новом тысячелетии, когда символические имена имеют одинаковое значение по всемирным компьютерным сетям, быстрые алгоритмы поиска играли и продолжают играть важную роль в вычислениях. В числе современных применений таблиц имен — организация научных данных, от поиска маркеров или повторяющихся последовательностей в генетических данных до карт всей вселенной; организация знаний в веб, от поиска в онлайновом магазине до переноса в сеть целых библиотек; и реализация инфраструктуры Интернета, от маршрутизации пакетов между серверами до совместных файловых систем и потокового видео. Все это стало возможным благодаря этим и другим важным приложениям. В данном разделе мы рассмотрим несколько характерных примеров.

- Клиент словаря и клиент индексации, позволяющие быстрый и гибкий доступ к информации в файлах с данными, разделяемыми запятыми (и в аналогичных форматах), которые часто применяются для хранения данных в веб-сети.
- Клиент индексации для построения инвертированного индекса для набора файлов.
- Тип данных для разреженной матрицы, где таблица имен используется для работы с задачами таких размеров, которые недоступны для обычной реализации.

В главе 6 мы рассмотрим таблицу символов, пригодную для организации баз данных и файловых систем, которые содержат сколь угодно огромное количество ключей.

Таблицы имен также играют важную роль в алгоритмах, которые еще будут рассматриваться далее в этой книге. Например, они позволяют представлять графы (глава 4) и обрабатывать строки (глава 5).

Как мы уже видели в данной главе, разработка реализаций для таблиц имен, которые могут гарантированно обеспечить быстрое выполнение всех операций, представляет собой серьезную и сложную задачу. Однако мы рассмотрели реализации, которые хорошо изучены, широко применяются и доступны во многих программных средах (в том числе и в библиотеках Java). Так что теперь вы, несомненно, будете рассматривать абстракцию таблицы имен как ключевой компонент в вашем инструментальном наборе программиста.

Так какую же реализацию таблицы имен лучше использовать?

Таблица 3.5.1 содержит сводку характеристик производительности для изученных в этой главе алгоритмов (кроме худших случаев для хеширования, которые взяты из исследовательской литературы и вряд ли встретятся на практике). В этой таблице видно, что в случае типичных приложений нам предстоит выбор между хеш-таблицами и деревьями бинарного поиска.

Таблица 3.5.1. Сводка асимптотических стоимостей для реализаций таблиц имен

Алгоритм (структура данных)	Стоимость в худшем случае (после N вставок)		Средняя стоимость (после N случайных вставок)		Интерфейс ключей	Затраты памяти (в байтах)
	поиск	вставка	попадание	вставка		
Последовательный поиск (неупорядоченный список)	N	N	$N/2$	N	`equals()`	$48N$
Бинарный поиск (упорядоченный массив)	$\lg N$	N	$\lg N$	$N/2$	`compareTo()`	$16N$
Поиск в бинарном дереве (ДБП)	N	N	$1{,}39 \lg N$	$1{,}39 \lg N$	`compareTo()`	$64N$
Поиск в 2-3-дереве (красно-черное ДБП)	$2\lg N$	$2\lg N$	$1{,}00 \lg N$	$1{,}00 \lg N$	`compareTo()`	$64N$
Раздельные цепочки* (массив списков)	$< \lg N$	$< \lg N$	$N/(2M)$	N/M	`equals()` `hashCode()`	$48N+64M$
Линейное опробование* (параллельные массивы)	$c\lg N$	$c\lg N$	$< 1{,}50$	$< 2{,}50$	`equals()` `hashCode()`	от $32N$ до $128N$

**Если хеш-функция дает равномерное и независимое распределение*

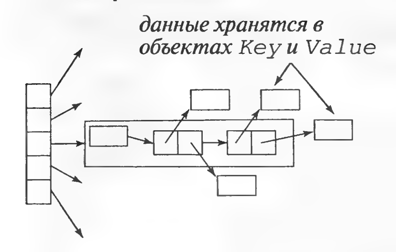
Хеширование лучше реализаций ДБП тем, что его код проще, а время поиска оптимально (константно) — если ключи имеют стандартный тип или достаточно просты для разработки эффективной хеш-функции, которая (приблизительно) удовлетворяет предположению о равномерном хешировании. А ДБП лучше хеширования тем, что они основаны на более простом абстрактном интерфейсе (нет необходимости сочинять хеш-функцию); красно-черные ДБП гарантируют хорошую производительность в худшем случае; и они поддерживают более широкий спектр операций (ранг, выбор, сортировка и поиск диапазона). Как правило, программисты используют хеширование всегда, кроме случаев, когда важен один или несколько таких факторов, для которых и были созданы красно-черные ДБП. В главе 5 мы рассмотрим одно исключение из этого правила: если ключи представляют собой длинные строки, то можно создать структуры данных, еще более гибкие по сравнению с красно-черными ДБП и даже более быстрые, чем хеширование.

Наши реализации таблиц имен пригодны для широкого спектра приложений, но описанные алгоритмы легко приспособить и к ряду других ситуаций, которые часто встречаются и достойны внимательного изучения.

Примитивные типы

Допустим, у нас имеется таблица имен с целочисленными ключами и связанными с ними числами с плавающей точкой. При использовании нашего стандартного подхода ключи и значения хранятся в значениях типов-оболочек `Integer` и `Double`, и для обращения к каждой паре ключ-значение нужны две дополнительных ссылки. Эти ссылки могут быть вполне приемлемы в приложении, где выполняются тысячи поисков по тысячам ключей, но могут оказаться серьезной лишней нагрузкой в приложении с миллиардами поисков по миллионам ключей. Использование примитивного типа вместо `Key` сэкономит одну ссылку на пару ключ-значение. Если связанное значение тоже примитивно, можно устранить и другую ссылку. Такая ситуация для раздельных цепочек показана на рис. 3.5.1; аналогичные компромиссы действуют и для других реализаций. Для приложений, где важна производительность, хорошо (и обычно несложно) разработать

Обычная реализация



Реализация примитивными типами

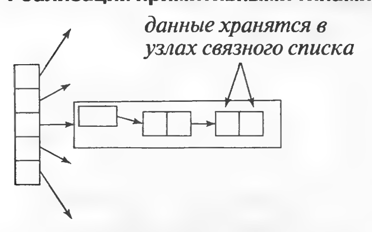


Рис. 3.5.1. Затраты памяти для раздельных цепочек

"сокращенные" версии наших реализаций (см. упражнение 3.5.4).

Повторяющиеся ключи

Возможность повторения ключей иногда требует специальной обработки в реализациях таблиц имен. Во многих приложениях желательно связывать с одним ключом несколько значений. Например, в системе обработки транзакций несколько транзакций могут иметь одинаковое значение кода пользователя. У нас принято соглашение о запрете одинаковых ключей, и управление такими ситуациями возлагается на клиент. Ниже в данном разделе мы рассмотрим пример такого клиента. Многие наши реализации будут иметь возможность оставлять пары ключ-значение с одинаковыми ключами в первичной структуре поиска данных и возвращать *любое* значение с указанным ключом в результате поиска. Можно добавить и методы для возврата *всех* значений с указанным ключом. Наши реализации ДБП и хеширования несложно приспособить к хранению повторяющихся ключей в структуре данных, однако для красно-черных ДБП это сделать немного сложнее (см. упражнения 3.5.9 и 3.5.10). Такие реализации часто приводятся в литературе (включая и предыдущие издания данной книги).

Библиотеки Java

Библиотеки `java.util.TreeMap` и `java.util.HashMap` представляют собой реализации таблиц имен на основе красно-черных ДБП и хеширования с раздельными цепочками соответственно. Библиотека `TreeMap` не поддерживает непосредственно методы `rank()`, `select()` и другие операции из нашего API упорядоченной таблицы имен, но их можно эффективно реализовать на основе поддерживаемых операций. Библиотека `HashMap` примерно эквивалентна нашей реализации `LinearProbingST`: в ней используется изменение размера для поддержания коэффициента загрузки порядка 75%.

Для согласованности и конкретности мы в данной книге будем использовать реализацию таблицы имен на основе ДБП из раздела 3.3 или на основе хеширования с линейным опробованием из раздела 3.4. Для краткости и чтобы подчеркнуть независимость клиента от реализации, мы будем использовать в клиентском коде имя `ST` в качестве сокращения вместо `RedBlackBST` для упорядоченной таблицы имен и `HashST` в качестве сокращения вместо `LinearProbingHashST`, если порядок не важен и доступны нужные хеш-функции. При этом мы будем понимать, что некоторые приложения могут иметь особые запросы, которые могут потребовать изменения или расширения одного из этих алгоритмов или структур данных. И, какую бы таблицу имен вы ни выбрали, протестируйте свой выбор, чтобы убедиться, что ее производительность вас устраивает.

API множеств

Некоторым клиентам таблиц имен не нужны значения, им достаточно возможности вставлять ключи в таблицу и проверять, находится ли ключ в таблице. Поскольку повторяющиеся ключи запрещены, эти операции соответствуют API, который предназначен для работы с *множеством* ключей в таблице, без соответствующих им значений (рис. 3.5.2).

public class **SET**<Key>		
	SET()	*создание пустого множества*
void	add(Key key)	*добавление ключа* key *в множество*
void	delete(Key key)	*удаление ключа* key *из множества*
boolean	contains(Key key)	*находится ли* key *в множестве?*
boolean	isEmpty()	*пусто ли множество?*
int	size()	*количество ключей в множестве*
Stringto	**String()**	*строковое представление множества*

Рис. 3.5.2. *API базового типа данных для множества*

Любую реализацию таблицы имен можно превратить в реализацию SET, игнорируя значения или используя простой класс-оболочку (см. упражнения 3.5.1–3.5.3).

Чтобы расширить тип SET и включить в него операции *объединение*, *пересечение*, *дополнение* и другие распространенные математические операции, нужен более сложный API (например, для операции *дополнение* нужен какой-то механизм задания *полного множества* всех возможных ключей), а также потребуется решить ряд интересных алгоритмических задач (см. упражнение 3.5.17).

Как и в случае типа ST, могут быть неупорядоченная и упорядоченная версии SET. Если ключи имеют тип Comparable, в API можно включить методы min(), max(), floor(), ceiling(), deleteMin(), deleteMax(), rank(), select() и варианты методов size() и get() с двумя аргументами и получить полный API для упорядоченных ключей. В соответствии с нашим соглашением по типу ST, мы будем использовать в клиентском коде имя SET для упорядоченных множеств и имя HashSET, если порядок ключей не важен.

Для демонстрации применения типа SET мы рассмотрим клиенты *фильтра*, которые читают последовательность строк из стандартного ввода и выводят некоторые из них в стандартный вывод. Такие клиенты применялись раньше в вычислительных системах, где оперативная память была слишком мала и не могла содержать все данные. Они не утратили своего значения и сейчас, когда программы берут входные данные из веб. В качестве демонстрационного набора входных данных мы будем использовать файл tinyTale.txt (см. листинг 3.1.2). Для наглядности в наших примерах мы переносим в выходные данные и символы новой строки, хотя в коде это не делается.

Дедупликация

Характерным примером фильтра является клиент класса SET или HashSET, который удаляет дубликаты из входного потока. Эта операция часто называется *дедупликацией* (см. листинг 3.5.1). Для нее используется множество просмотренных к текущему моменту ключей. Если очередной ключ *присутствует* внутри множества, его нужно проигнорировать, а если он *отсутствует*, то нужно добавить его в множество и вывести в выходной поток. Ключи выводятся в стандартный вывод в порядке их чтения из стандартного ввода, но без повторений. Этому процессу нужен объем памяти, пропорциональный количеству различных ключей во входном потоке (который обычно значительно меньше общего количества ключей).

Листинг 3.5.1. Фильтр дедупликации

```
public class DeDup
{
   public static void main(String[] args)
   {
      HashSET<String> set;
      set = new HashSET<String>();
      while (!StdIn.isEmpty())
      {
         String key = StdIn.readString();
         if (!set.contains(key))
         {
            set.add(key);
            StdOut.println(key);
         }
      }
   }
}
```

```
% java DeDup < tinyTale.txt
it was the best of times worst
age wisdom foolishness
epoch belief incredulity
season light darkness
spring hope winter despair
```

Белый список и черный список

Другой классический фильтр использует ключи из отдельного файла для определения, которые ключи из входного потока нужно пропускать в выходной поток. Этот общий процесс имеет много естественных применений. Самый простой пример — это *белый список*, когда любой ключ, присутствующий в файле, считается "хорошим". Один клиент может пропускать в стандартный вывод все ключи, которые *отсутствуют* в белом списке, и игнорировать все ключи, которые *присутствуют* в нем (как в нашей первой программе в главе 1); а другой клиент может пропускать в стандартный вывод все ключи, которые *присутствуют* в белом списке, и игнорировать все ключи, которые *отсутствуют* в нем (как показано в клиенте `WhiteFilter` класса `HashSET` в листинге 3.5.2). Например, почтовое приложение может использовать такой фильтр для указания адресов друзей, почту от которых надо принимать, а все остальные сообщения считать спамом. Для этого нужно создать множество `HashSET` ключей из указанного списка, а потом читать ключи из стандартного ввода. Если очередной ключ *присутствует* внутри множества, он выводится, а если *отсутствует*, то игнорируется.

Листинг 3.5.2. Фильтр белого списка

```
public class WhiteFilter
{
   public static void main(String[] args)
   {
      HashSET<String> set;
      set = new HashSET<String>();
```

```
    In in = new In(args[0]);
    while (!in.isEmpty())
      set.add(in.readString());
    while (!StdIn.isEmpty())
    {
      String word = StdIn.readString();
      if (set.contains(word))
        StdOut.println(word);
    }
  }
}
```

```
% more list.txt
was it the of

% java WhiteFilter list.txt < tinyTale.txt
it was the of it was the of
it was the of it was the of
it was the of it was the of
it was the of it was the of
it was the of it was the of

% java BlackFilter list.txt < tinyTale.txt
best times worst times
age wisdom age foolishness
epoch belief epoch incredulity
season light season darkness
spring hope winter despair
```

Черный список действует наоборот: любой ключа, который присутствует в файле, считается "плохим". Для клиентов черного списка также возможны два естественных фильтра. В нашем почтовом примере можно задать адреса известных спамеров и указать почтовому приложению пропускать всю почту, кроме сообщений с любого из этих адресов. Клиент `BlackFilter` класса `HashSET` можно реализовать, инвертировав результат теста проверки в `WhiteFilter`.

Типичные практические ситуации — банк, выпускающий кредитные карты, которому нужно отфильтровывать номера украденных и утерянных карт, или Интернет-маршрутизатор с белым списком для реализации брандмауэра. В таких приложениях могут быть огромные списки, неограниченные входные потоки и жесткие требования к времени отклика. Реализации таблиц имен, подобные рассмотренным в данной главе, позволяют легко удовлетворить таким требованиям.

Клиенты словарей

Наиболее характерный вариант клиента таблицы имен создает таблицу имен с помощью последовательности операций *занести*, чтобы затем отвечать на запросы *выбрать*. Во многих приложениях используются таблицы имен, которые выполняют функции *динамического* словаря, где легко не только найти информацию в таблице, *но и* изменять эту информацию.

Пользу этого подхода демонстрирует следующий список знакомых примеров.

- *Телефонный справочник.* Если ключами являются имена людей, а значениями — номера их телефонов, то таблица имен моделирует телефонный справочник. Такой справочник существенно отличается от типографского тем, что в него можно добавлять новые имена или изменять существующие номера телефонов. Можно и наоборот — использовать номера телефонов в качестве ключей, а имена в качестве значений; если вы еще так не делали, попробуйте набрать свой номер телефона (с международным кодом) в поле поиска своего браузера.
- *Словарь.* Привязка к слову его толкования — знакомая всем концепция "словаря". Веками дома и в офисах люди пользовались печатными словарями, чтобы посмотреть определения и правописание (значения) слов (ключей). Но теперь, при наличии хороших реализаций таблиц имен, в основном люди пользуются средствами проверки орфографии, встроенными в текстовые процессоры, и могут немедленно получить определения слов на своих компьютерах (табл. 3.5.2).
- *Биржевая информация.* Люди, имеющие акции каких-то предприятий, регулярно просматривают их цены в сети. Различные службы связывают символ тикера (ключ) с текущей ценой (значение), обычно вместе с другой информацией. Подобных коммерческих приложений множество — например, финансовые учреждения, связывающие номер счета с его названием или образовательные учреждения, связывающие оценки студента с его именем или идентификационным номером.
- *Геномика.* Символы играют ключевую роль в современной геномике. Простейший пример — представление буквами `A`, `C`, `T` и `G` нуклеотидов, присутствующих в ДНК живых организмов. Чуть более сложным примером является соответствие между кодонами (тройки нуклеотидов) и аминокислотами (`TTA` соответствует лейцину, `TCT` — серину и т.д.), еще более сложным — соответствие между последовательностями аминокислот и протеинами, и т.д. Исследователи в области геномики часто применяют различные виды таблиц имен для организации своих сведений.
- *Экспериментальные данные.* Современные ученые работают с огромными объемами экспериментальных данных, и организация и эффективный доступ к этим данным просто необходим для их анализа. Таблицы имен являются критической отправной точкой, и сложные структуры данных и алгоритмы, основанные на таблицах имен, представляют собой сейчас важный аспект научных исследований.
- *Компиляторы.* Одним из самых первых применений таблиц имен была организация информации для программирования. Вначале программы были просто последовательностью чисел, но программисты очень быстро поняли, что для операций и местоположений в памяти гораздо удобнее использовать символические имена (имена переменных). А для связи имен с числами были нужны таблицы имен. По мере роста размера программ стоимость операций в таблицах имен стала узким местом в цикле разработки программ. Это привело к разработке структур данных и алгоритмов, подобных описанным в данной главе.
- *Файловые системы.* Таблицы имен постоянно применяются для организации данных в вычислительных системах. Пожалуй, наиболее характерным примером является *файловая система*, где имя файла (ключ) связывается с местоположением его содержимого (значение). Музыкальный плеер использует такую же систему для связи названий песен (ключи) с местоположением самих записей (значение).

- *DNS в Интернете.* Система доменных имен (domain name system — DNS) представляет собой основу организации информации в Интернете и связывает URL-адреса (ключи), понятные людям (вроде www.princeton.edu или www.wikipedia.org), с IP-адресами (значения), понятными сетевым маршрутизаторам (наподобие 208.216.181.15 или 207.142.131.206). То есть это как бы "телефонная книга" следующего поколения. Поэтому пользователи могут использовать легко запоминаемые имена, а машины могут эффективно работать с числами. Количество поисков в таблицах имен, выполняемых для этого каждую секунду в Интернет-маршрутизаторах, настолько велико, что производительность очевидным образом выходит на передний план. Ежегодно к Интернету подключаются миллионы новых компьютеров и других устройств, поэтому эти таблицы имен должны быть динамическими.

Таблица 3.5.2. Типичные применения словарей

Область	Ключ	Значение
Телефонная книга	Имя	Номер телефона
Словарь	Слово	Определение
Бухгалтерия	Номер счета	Баланс
Геномика	Кодон	Аминокислота
Данные	Дата/время	Результаты
Компиляторы	Имя переменной	Местоположение в памяти
Файловые системы	Название песни	Местоположение на диске
DNS	Веб-сайт	IP-адрес

Несмотря на значительный объем, этот список — всего лишь репрезентативная выборка, которая демонстрирует область применимости абстракции таблицы имен. Если можно что-то назвать по имени, то можно применять и таблицу имен. Файловая система компьютера или DNS-серверы могут выполнить за вас работу, но за ней стоит скрытая где-то таблица имен.

В качестве конкретного примера мы рассмотрим клиент таблицы имен для поиска информации, которая хранится в таблице — в файле или на веб-странице — с использованием формата *данных, разделяемых запятыми* (.csv). Этот простой формат служит (скромной) цели хранения табличных данных в виде, который доступен (и будет доступен) для чтения без специальных приложений: данные записаны в текстовом виде, по одной записи в строке, и элементы записей разделены запятыми (см. листинг 3.5.3). На сайте книги содержатся много .csv-файлов, предназначенные для различных перечисленных нами приложений: amino.csv (кодировки кодонов аминокислотами), DJIA.csv (цена открытия, объем и цена закрытия индекса Доу-Джонса, по дням), ip.csv (выборка элементов из базы данных DNS) и upc.csv (единые штрих-коды, широко применяемые для идентификации продуктов потребления). Электронные таблицы и другие приложения обработки данных могут читать файлы в формате .csv, и наш пример демонстрирует, что можно написать Java-программу обработки таких данных любым нужным способом.

Листинг 3.5.3. Типичные файлы с разделением данных запятой (.csv)

```
% more amino.csv
TTT,Phe,F,Phenylalanine
TTC,Phe,F,Phenylalanine
TTA,Leu,L,Leucine
TTG,Leu,L,Leucine
TCT,Ser,S,Serine
TCC,Ser,S,Serine
...
GAA,Gly,G,Glutamic Acid
GAG,Gly,G,Glutamic Acid
GGT,Gly,G,Glycine
GGC,Gly,G,Glycine
GGA,Gly,G,Glycine
GGG,Gly,G,Glycine
% more DJIA.csv
...
20-Oct-87,1738.74,608099968,1841.01
19-Oct-87,2164.16,604300032,1738.74
16-Oct-87,2355.09,338500000,2246.73
15-Oct-87,2412.70,263200000,2355.09
...
30-Oct-29,230.98,10730000,258.47
29-Oct-29,252.38,16410000,230.07
28-Oct-29,295.18,9210000,260.64
25-Oct-29,299.47,5920000,301.22
...
% more ip.csv
...
www.ebay.com,66.135.192.87
www.princeton.edu,128.112.128.15
www.cs.princeton.edu,128.112.136.35
www.harvard.edu,128.103.60.24
www.yale.edu,130.132.51.8
www.cnn.com,64.236.16.20
www.google.com,216.239.41.99
www.nytimes.com,199.239.136.200
www.apple.com,17.112.152.32
www.slashdot.org,66.35.250.151
www.espn.com,199.181.135.201
www.weather.com,63.111.66.11
www.yahoo.com,216.109.118.65
...
% more UPC.csv
...
0002058102040,,"1 1/4"" STANDARD STORM DOOR"
0002058102057,,"1 1/4"" STANDARD STORM DOOR"
0002058102125,,"DELUXE STORM DOOR UNIT"
0002082012728,"100/ per box","12 gauge shells"
0002083110812,"Classical CD","'Bits and Pieces'"
002083142882,CD,"Garth Brooks - Ropin' The Wind"
0002094000003,LB,"PATE PARISIEN"
0002098000009,LB,"PATE TRUFFLE COGNAC-M&H 8Z RW"
0002100001086,"16 oz","Kraft Parmesan"
0002100002090,"15 pieces","Wrigley's Gum"
0002100002434,"One pint","Trader Joe's milk"
...
```

Программа LookupCSV (листинг 3.5.4) создает пары ключ-значение из файла разделенных запятыми значений, который указывается в командной строке, а затем выводит значения, соответствующие ключам из стандартного ввода. Аргументами командной строки являются имя файла и два целых числа: одно указывает поле, служащее ключом, а другое — поле, служащее значением.

Листинг 3.5.4. Поиск в словаре

```
public class  LookupCSV
{
  public static void main(String[] args)
  {
    In in = new In(args[0]);
    int keyField = Integer.parseInt(args[1]);
    int valField = Integer.parseInt(args[2]);

    ST<String, String> st = new ST<String, String>();

    while (in.hasNextLine())
    {
      String line = in.readLine();
      String[] tokens = line.split(",");
      String key = tokens[keyField];
      String val = tokens[valField];
      st.put(key, val);
    }
    while (!StdIn.isEmpty())
    {
      String query = StdIn.readString();
      if (st.contains(query))
        StdOut.println(st.get(query));
    }
  }
}
```

Этот управляемый данными клиент таблицы имен читает из файла пары ключ-значение, а затем выводит значения, соответствующие ключам из стандартного ввода. И ключи, и значения представляют собой строки. Разграничивающий символ вводится как аргумент командной строки.

```
% java LookupCSV ip.csv 1 0
128.112.136.35
www.cs.princeton.edu

% java LookupCSV DJIA.csv 0 3
29-Oct-29
230.07

% java LookupCSV amino.csv 0 3
TCC
Serine

% java LookupCSV UPC.csv 0 2
0002100001086
Kraft Parmesan
```

Этот пример иллюстрирует пользу и гибкость абстракции таблицы имен. У какого веб-сайта IP-адрес 128.112.136.35? Ответ: у www.cs.princeton.edu. Какая аминокислота соответствует кодону TCA? — Ответ: серин (Serine). Каков был индекс Доу-Джонса 29 октября 1929 г.? Ответ: 252.38. У какого продукта код 0002100001086? Ответ: у Kraft Parmesan. Клиент LookupCSV, которому передан соответствующий .csv-файл, с легкостью может ответить на подобные вопросы.

При обработке интерактивных запросов производительность не очень важна (т.к. за время ввода запроса компьютер может просмотреть миллионы элементов), поэтому при работе с LookupCSV скорость работы класса ST трудно оценить. Но если запросы отправляет *программа* (да еще и огромное количество), то тут уж производительность важна. Например, Интернет-маршрутизатору может понадобиться выполнять миллионы поисков IP-адресов. В этой книге мы уже видели, насколько важной бывает хорошая производительность (FrequencyCounter), и в данном разделе мы познакомимся с еще несколькими примерами.

Примеры похожих, но более сложных тестовых клиентов для .csv-файлов приведены в упражнениях. Например, словарь можно сделать динамическим, разрешив присутствие в стандартном вводе команд изменения значения, связанного с ключом, или команд поиска диапазона, либо можно строить из одного и того же файла несколько словарей.

Клиенты индексации

Для словарей характерно свойство, что с каждым ключом связано одно значение, поэтому для них удобно непосредственно применять наш тип данных ST, основанный на абстракции ассоциативного массива, который назначает каждому ключу одно значение. Каждый номер счета уникально идентифицирует клиента, каждый штрих код уникально идентифицирует товар и т.д. Конечно, в общем случае с каждым ключом могут быть связаны несколько значений. Например, в нашем примере amino.csv каждый кодон идентифицирует одну аминокислоту, однако каждая аминокислота связана со списков кодонов, как в файле aminoI.csv (рис. 3.5.3), где каждая строка содержит одну аминокислоту и список связанных с ней кодонов. Для названия таблиц имен с несколькими значениями у каждого ключа мы используем термин *индекс* (табл. 3.5.3). Ниже перечислены еще примеры.

aminoI.txt

```
Alanine,AAT,AAC,GCT,GCC,GCA,GCG
Arginine,CGT,CGC,CGA,CGG,AGA,AGG
Aspartic Acid,GAT,GAC
Cysteine,TGT,TGC
Glutamic Acid,GAA,GAG
Glutamine,CAA,CAG
Glycine,GGT,GGC,GGA,GGG
Histidine,CAT,CAC
Isoleucine,ATT,ATC,ATA
Leucine,TTA,TTG,CTT,CTC,CTA,CTG
Lysine,AAA,AAG
Methionine,ATG
Phenylalanine,TTT,TTC
Proline,CCT,CCC,CCA,CCG
Serine,TCT,TCA,TCG,AGT,AGC
Stop,TAA,TAG,TGA
Threonine,ACT,ACC,ACA,ACG
Tyrosine,TAT,TAC
Tryptophan,TGG
Valine,GTT,GTC,GTA,GTG
```

разделяющая запятая

ключ *значения*

Рис. 3.5.3. *Небольшой индексный файл (20 строк)*

- *Коммерческие транзакции.* Один из способов, каким компания может учитывать клиентские счета — это хранение транзакций, выполненных на протяжении дня, в индексе дневных транзакций. Ключом служит номер счета, а значение — список транзакций, где участвовал этот счет.

- *Поиск в веб-сети.* Когда вы вводите слово и получаете список веб-сайтов, содержащих это слово, вы пользуетесь индексом, созданным поисковым механизмом. С каждым ключом (запрос) связано одно значение (множество страниц), хотя в реальности часто можно задавать несколько ключей.
- *Фильмы и актеры.* Файл `movies.txt` на сайте книги (небольшая его часть приведена на рис. 3.5.4) взят из Интернет-базы данных фильмов (Internet Movie Database — IMDB). Каждая строка содержит название фильма (ключ), а за ним — список актеров в этом фильме (значение), разделенных слешами.

```
movies.txt
  ...
  Tin Men (1987)/DeBoy, David/Blumenfeld, Alan/...
  Tirez sur le pianiste (1960)/Heymann, Claude/...
  Titanic (1997)/Mazin, Stan/...DiCaprio, Leonardo/...
  Titus (1999)/Weisskopf, Hermann/Rhys, Matthew/...
  To Be or Not to Be (1942)/Verebes, Ernö (I)/...
  To Be or Not to Be (1983)/.../Brooks, Mel (I)/...
  To Catch a Thief (1955)/París, Manuel/...
  To Die For (1995)/Smith, Kurtwood/.../Kidman, Nicole/...
  ...
```


Рис. 3.5.4. Небольшая часть крупного индексного файла (более 250 000 строк)

Таблица 3.5.3. Типичные применения индексации

Область	Ключ	Значение
Геномика	Аминокислота	Список кодонов
Финансы	Номер счета	Список транзакций
Веб-поиск	Ключ поиска	Список веб-страниц
IMDB	Фильм	Список актеров

Индекс несложно построить, помещая значения, которые надо связать с каждым ключом, в одну структуру данных (например, в `Queue`), а затем надо связать ключ с этой структурой данных, как со значением. Расширить класс `LookupCSV` в соответствии с этим принципом совсем не сложно, и мы оставляем такое расширение на самостоятельную проработку (см. упражнение 3.5.12). Вместо этого мы рассмотрим класс `LookupIndex` (листинг 3.5.5), который строит таблицу имен для создания индекса из файлов, подобных `aminoI.txt` и `movies.txt`, где символ-разделитель может быть не только запятой, и его можно указать в командной строке. После создания индекса объект `LookupIndex` может принимать запросы с ключами и выводить значения, связанные с каждым ключом. Но интереснее то, что `LookupIndex` строит для каждого файла и *инвертированный индекс*, где значения и ключи меняются ролями. В примере с аминокислотами он будет работать, как `Lookup` (поиск аминокислоты, связанной с указанным кодоном), в примере с киноактерами можно будет найти фильмы, в которых принимал участие указанный актер — все эти данные есть в файлах, но их трудно получить без таблиц имен. *Внимательно изучите данный пример*: он позволит лучше понять суть таблиц имен.

15 Зак. 3239

Листинг 3.5.5. Поиск по индексу (и инвертированному индексу)

```
public class  LookupIndex
{
  public static void main(String[] args)
  {
    In in = new In(args[0]);          // база данных для индекса
    String sp = args[1];              // разделитель

    ST<String, Queue<String>> st = new ST<String, Queue<String>>();
    ST<String, Queue<String>> ts = new ST<String, Queue<String>>();

    while (in.hasNextLine())
    {
      String[] a = in.readLine().split(sp);
      String key = a[0];
      for (int i = 1; i < a.length; i++)
      {
        String val = a[i];
        if (!st.contains(key)) st.put(key, new Queue<String>());
        if (!ts.contains(val)) ts.put(val, new Queue<String>());
        st.get(key).enqueue(val);
        ts.get(val).enqueue(key);
      }
    }

    while (!StdIn.isEmpty())
    {
      String query = StdIn.readLine();
      if (st.contains(query))
        for (String s : st.get(query))
          StdOut.println("  " + s);
      if (ts.contains(query))
        for (String s : ts.get(query))
          StdOut.println("  " + s);
    }
  }
}
```

Этот управляемый данными клиент таблицы имен читает из файла пары ключ-значение, а затем выводит значения, соответствующие ключам из стандартного ввода. Ключи представляют собой строки, а значения — списки строк. Разграничивающий символ вводится как аргумент командной строки.

```
% java LookupIndex aminoI.txt ","
Serine
  TCT
  TCA
  TCG
  AGT
  AGC
  TCG
  Serine
```

```
% java LookupIndex movies.txt "/"
Bacon, Kevin
  Mystic River (2003)
  Friday the 13th (1980)
  Flatliners (1990)
  Few Good Men, A (1992)
  ...
Tin Men (1987)
  Blumenfeld, Alan
  DeBoy, David
  ...
```

Инвертированный индекс

Термин *инвертированный* (или *обратный*) *индекс* обычно применяется, когда по значениям ищутся *ключи*. Имеется большой объем данных, и требуется узнать, где присутствуют нужные нам ключи (табл. 3.5.4). Это приложение — еще один типичный пример клиента таблицы имен со смешанной последовательностью вызовов `get()` и `put()`. Здесь также каждый ключ связывается с множеством мест, где этот ключ можно найти. Способ использования этих мест зависит от приложения: в книге это могут быть номера страниц, в программе — номера строк, в геномике — позиции генетических последовательностей и т.д.

- *Интернет-база данных фильмов* (Internet Movie Database — IMDB). В недавно рассмотренном примере входными данными является индекс, связывающий каждый фильм со списком актеров. Инвертированный индекс связывает каждого актера со списком фильмов.
- *Индекс в книге.* В каждом учебнике есть индекс, где можно найти термин, а для него — номера страниц, содержащих этот термин. При создании хорошего индекса автор книги обычно не включает в него распространенные и не относящиеся к теме слова, и система подготовки документов наверняка применяет таблицу имен, чтобы автоматизировать этот процесс. Интересный специальный случай называется *алфавитным указателем* (конкордация, или соответствие). Он связывает каждое слово из текста с множеством позиций в тексте, где это слово встречается (см. упражнение 3.5.20).
- *Компилятор.* В крупной программе с большим количеством переменных полезно знать, где используется каждая переменная. Исторически явно напечатанная таблица имен была одним из наиболее важных средств, которое позволяло увидеть, где используются переменные в программе. В современных системах таблицы имен лежат в основе программных средств, применяемых для управления названиями программ.
- *Поиск в файлах.* В современных операционных системах можно напечатать фрагмент текста и получить список файлов, где встречается этот фрагмент. Ключом является фрагмент, а значением — множество файлов, содержащих этот фрагмент.
- *Геномика.* В практической (хотя и упрощенной) ситуации в поиске, используемом в геномике, ученому необходимо найти позиции некоторой генетической последовательность в существующем геноме или наборе геномов. Существование точного или приблизительного соответствия может иметь важное научное значение. Основой для такого поиска является индекс вроде алфавитного указателя, но с учетом того, что геномы не поделены на слова (см. упражнение 3.5.15).

Таблица 3.5.4. Типичные инвертированные индексы

Область	Ключ	Значение
IMDB	Актер	Набор фильмов
Книга	Термин	Набор страниц
Компилятор	Идентификатор	Набор мест в программе
Поиск в файлах	Искомый фрагмент	Набор файлов
Геномика	Подпоследовательность	Набор местоположений

Класс `FileIndex` (см. листинг 3.5.6) принимает в командной строке имена файлов и с помощью таблицы имен строит инвертированный индекс, связывающий каждое слово во всех файлах с множеством имен файлов, где присутствует это слово. Затем он читает из стандартного ввода запросы с ключевыми словами и формирует соответствующий список файлов. Этот процесс похож на привычный поиск информации в веб: вы вводите ключевое слово и получаете список мест, где это слово встречается. Разработчики таких средств обычно добавляют в них дополнительные возможности, обращая тщательное внимание на следующие моменты:

- форма запроса;
- множество индексируемых файлов/страниц;
- порядок, в котором файлы перечислены в ответе.

Листинг 3.5.6. Индексация файла

```
import java.io.File;
public class FileIndex
{
   public static void main(String[] args)
   {
      ST<String, SET<File>> st = new ST<String, SET<File>>();
      for (String filename : args)
      {
         File file = new File(filename);
         In in = new In(file);
         while (!in.isEmpty())
         {
            String word = in.readString();
            if (!st.contains(word)) st.put(word, new SET<File>());
            SET<File> set = st.get(word);
            set.add(file);
         }
      }
      while (!StdIn.isEmpty())
      {
         String query = StdIn.readString();
         if (st.contains(query))
            for (File file : st.get(query))
               StdOut.println("  " + file.getName());
      }
   }
}
```

Этот клиент таблицы имен индексирует набор файлов. Для каждого слова из каждого файла выполняется поиск в таблице имен, содержащей множество имен файлов, в которых встречается данное слово. Имена для In могут также относиться к веб-страницам, поэтому данный код можно использовать для создания инвертированного индекса веб-страниц.

```
% more ex1.txt
it was the best of times

% more ex2.txt
it was the worst of times

% more ex3.txt
it was the age of wisdom

% more ex4.txt
it was the age of foolishness

% java FileIndex ex*.txt
age
  ex3.txt
  ex4.txt
best
  ex1.txt
was
  ex1.txt
  ex2.txt
  ex3.txt
  ex4.txt
```

Например, вы наверняка часто печатали поисковые запросы в Интернете, содержащие несколько слов (для этого выполняется индексации больших частей страниц), ответы на которые упорядочиваются по релевантности или важности (для пользователей или рекламодателей). Упражнения в конце данного раздела касаются некоторых из таких усовершенствований. Различные алгоритмические вопросы, связанные с веб-поиском, мы рассмотрим позже, но таблица имен наверняка лежит в основании таких процессов.

Как и в случае LookupIndex, мы рекомендуем загрузить исходный код класса FileIndex с сайта книги и использовать его для индексации каких-нибудь текстовых файлов на вашем компьютере или на интересующих вас веб-сайтах — так вы еще больше ощутите пользу таблиц имен. Если вы сделаете так, то вы увидите, что данный класс может создавать большие индексы для огромных файлов за небольшое время, поскольку все операции *занести* и *выбрать* выполняются немедленно. Обеспечение такого непосредственного отклика для огромных динамических таблиц — один из примеров триумфа классической технологии алгоритмов.

Разреженные векторы

Наш следующий пример демонстрирует важность таблиц имен в научных и математических вычислениях. Мы опишем фундаментальное и всем знакомое вычисление, которое часто становится узким местом в некоторых приложениях, а затем покажем, как

применение таблицы имен может расширить это узкое место и позволить решать задачи гораздо большего размера. На вычислении, которое мы рассмотрим, был основан алгоритм PageRank, который был разработан С. Брином и Л. Пейджем в начале 2000-х годов и привел к возникновению поисковой системы Google (эта известная математическая абстракция полезна и во многих других контекстах).

Основное вычисление, которое мы рассмотрим — это *умножение матрицы на вектор*: для заданной матрицы и вектора нужно вычислить результирующий вектор, i-й элемент которого является *скалярным произведением* исходного вектора и i-й строки матрицы (рис. 3.5.5 и листинг 3.5.7). Для простоты мы рассмотрим случай квадратной матрицы с N строками и N столбцами и вектора размером N. Такую операцию несложно закодировать в Java, и она требует времени, пропорционального N^2, чтобы выполнить N умножений для вычисления каждого из N элементов результирующего вектора — и это согласуется с требованием к объему памяти, пропорциональным N^2, для хранения матрицы.

$$\underset{\texttt{a[][]}}{\begin{bmatrix} 0 & .90 & 0 & 0 & 0 \\ 0 & 0 & .36 & .36 & .18 \\ 0 & 0 & 0 & .90 & 0 \\ .90 & 0 & 0 & 0 & 0 \\ .47 & 0 & .47 & 0 & 0 \end{bmatrix}} \underset{\texttt{x[]}}{\begin{bmatrix} .05 \\ .04 \\ .36 \\ .37 \\ .19 \end{bmatrix}} = \underset{\texttt{b[]}}{\begin{bmatrix} .036 \\ .297 \\ .333 \\ .045 \\ .1927 \end{bmatrix}}$$

Рис. 3.5.5. Умножение матрицы на вектор

Листинг 3.5.7. Стандартная реализация умножения матрицы на вектор

```
...
double[][] a = new double[N][N];
double[] x = new double[N];
double[] b = new double[N];
...
// Инициализация a[][] и x[].
...
for (int i = 0; i < N; i++)
{
   sum = 0.0;
   for (int j = 0; j < N; j++)
      sum += a[i][j]*x[j];
   b[i] = sum;
}
```

На практике значение N часто бывает очень большим. Например, в упоминавшемся выше приложении Google число N равно количеству страниц в веб. На момент разработки алгоритма PageRank это были десятки или сотни миллиардов, а с тех пор это число неимоверно возросло. Поэтому N^2 уже гораздо больше, чем 10^{20}. Никто не может выделить столько времени или памяти, и, значит, нужен лучший алгоритм.

К счастью, такие матрицы обычно *разреженные*: подавляющее большинство их элементов равны 0. И в приложении Google среднее количество ненулевых элементов в каждой строке равно небольшой константе: практически все веб-страницы содержат ссылки лишь на несколько других (а не на все страницы всемирной сети). Поэтому матрицу

можно представить в виде массива разреженных векторов (рис. 3.5.6), воспользовавшись реализацией SparseVector, которая приведена в листинге 3.5.8. Вместо кода a[i][j] для обращения к элементу в строке i и столбце j мы используем выражение a[i].put(j, val) для занесения значения в матрицу и a[i].get(j) для выборки значения. Как видно из листинга 3.5.9, умножение матрицы на вектор с помощью этого класса выполняется даже проще, чем с помощью "массивного" представления (да и вычисление описывается понятнее). Но более важно то, что для этого необходимо время, пропорциональное N плюс количество ненулевых элементов в матрице.

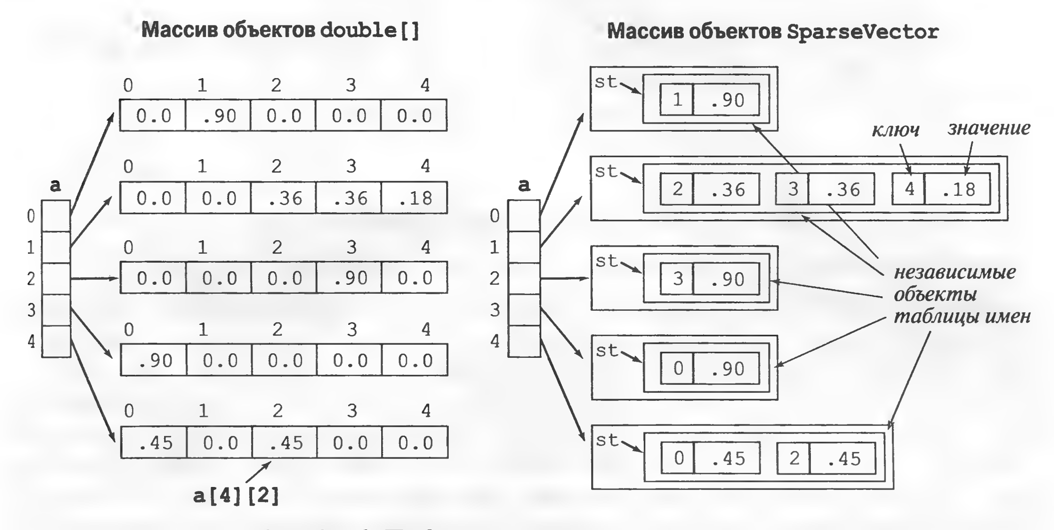


Рис. 3.5.6. Представления разреженных матриц

Листинг 3.5.8. Разреженный вектор и скалярное произведение

```
public class   SparseVector
{
  private HashST<Integer, Double> st;

  public SparseVector()
  {  st = new HashST<Integer, Double>();  }
  public int size()
  {  return st.size();  }
  public void put(int i, double x)
  {  st.put(i, x);  }
  public double get(int i)
  {
     if (!st.contains(i)) return 0.0;
     else return st.get(i);
  }
  public double dot(double[] that)
  {
     double sum = 0.0;
     for (int i : st.keys())
        sum += that[i]*this.get(i);
     return sum;
  }
}
```

Этот клиент таблицы имен — элементарная реализация разреженного вектора, которая демонстрирует эффективное выполнение скалярного перемножения разреженных векторов. Каждый элемент одного операнда умножается на соответствующий элемент другого операнда, и результат добавляется к текущей сумме. Количество умножений равно количеству ненулевых элементов в разреженном векторе.

Листинг 3.5.9. Умножение разреженной матрицы на вектор

```
...
SparseVector[] a;
a = new SparseVector[N];
double[] x = new double[N];
double[] b = new double[N];
...
// Инициализация a[] и x[].
...
for (int i = 0; i < N; i++)
   b[i] = a[i].dot(x);
```

Для небольших и не разреженных матриц затраты на хранение таблиц имен могут быть значительными, но не пожалейте времени, чтобы разобраться в этой разновидности применения таблиц имен для огромных разреженных матриц. При этом вспоминайте про какое-нибудь очень крупное приложение (вроде того, с которым пришлось работать Брину и Пейджу), где N равно 10 или 100 миллиардов, но среднее количество ненулевых элементов в каждой строке меньше 10. В таких случаях ***применение таблиц имен ускоряет умножение матрицы на вектор в миллиард раз и более***. С виду простая природа этого приложения не должна отвлекать вас от его важности: программисты, которые не используют возможности для подобной экономии времени и памяти, существенно ограничивают свой потенциал решения практических задач, а программисты, которые пользуются возможностями ускорения в миллиарды раз (если это возможно), могут решать задачи, непосильные другим способам.

Создание матрицы для Google — это приложение из области обработки графов (и клиент таблицы имен), несмотря на огромную разреженную матрицу. Для заданной матрицы вычисление по алгоритму PageRank сводится к умножению матрицы на вектор, замене исходного вектора полученным и повторению этого процесса до сходимости (которая гарантируется фундаментальными теоремами теории вероятностей). Поэтому использование класса вроде `SparseVector` может снизить затраты времени и памяти для такого приложения в 10–100 миллиардов раз и более.

Аналогичная экономия возможна во многих случаях, поэтому разреженные векторы и матрицы широко используются и обычно встроены в специализированные системы научных вычислений. При работе с огромными векторами и матрицами лучше прогонять специальные тесты производительности, чтобы удостовериться, что возможная экономия в действительности достигнута. Однако обработка массивов для примитивных типов данных встроена в большинство языков программирования, и поэтому использование массивов для не разреженных векторов (как в данном примере) также может сэкономить ресурсы. Чтобы принять соответствующее проектное решение для таких приложений, важно четко понимать природу всех затрат.

Таблицы имен — один из наиболее важных вкладов алгоритмической технологии в разработку нашей современной вычислительной инфраструктуры, поскольку они могут сэкономить значительные ресурсы в широком спектре практических приложений, а это означает разницу между решением множества задач и невозможностью даже подступиться к ним. Очень немногие области науки и техники изучают эффекты изобретений, которые снижают затраты в сотни миллиардов раз, а применения таблиц имен как раз из этого ряда, как мы убедились на нескольких примерах, и подобная экономия имеет ярко выраженных эффект. Несомненно, рассмотренные нами структуры данных и алгоритмы — не самое последнее слово: все они были разработаны на протяжении нескольких последних десятилетий, и их свойства еще не полностью изучены. В силу своей важности, реализации таблиц имен продолжают интенсивно изучаться исследователями во всем мире, и можно ожидать новых открытий на многих фронтах, по мере расширения масштаба соответствующих приложений.

Вопросы и ответы

Вопрос. **Может ли структура `SET` содержать значения `null`?**

Ответ. Нет. Как и в случае таблиц имен, ключи представляют собой ненулевые объекты.

Вопрос. **А может ли сама структура `SET` быть равна `null`?**

Ответ. Нет. Объект `SET` может быть пустым (не содержать объекты), но не нулевым. Как и в случае любого типа данных в Java, переменная типа `SET` может иметь значение `null`, но это просто означает, что она не указывает ни на какой объект типа `SET`. Операция `new` для создания экземпляра класса `SET` всегда дает ненулевой объект.

Вопрос. **Если все данные находятся в памяти, то какой смысл в использовании фильтра?**

Ответ. Правильно. Фильтрация полезна тогда, когда неизвестно, сколько данных можно ожидать. В остальных случаях это полезный способ мышления, но не панацея.

Вопрос. **Данные хранятся в электронной таблице. Можно ли разработать что-то вроде класса `LookupCSV` для поиска в этих данных?**

Ответ. Скорее всего, в табличном приложении имеется возможность экспорта данных в `.csv`-файл, поэтому класс `LookupCSV` можно использовать непосредственно.

Вопрос. **Зачем нужен класс `FileIndex`? Разве операционная система не выполняет те же функции?**

Ответ. Если вы работаете в среде ОС, которая удовлетворяет вашим потребностям, то, конечно, пользуйтесь ее средствами. Как и многие другие наши программы, цель `FileIndex` — продемонстрировать базовые механизмы таких приложений и предложить возможности для вас.

Вопрос. **А можно сделать так, чтобы метод `dot()` в классе `SparseVector` принимал в качестве аргумента объект `SparseVector` и возвращал объект `SparseVector`?**

Ответ. Это вполне удобный альтернативный подход и интересное упражнение по программированию, которое требует несколько более сложного кодирования, чем в нашем варианте (см. упражнение 3.5.16). В случае общей обработки матриц может оказаться полезным добавить также тип `SparseMatrix`.

Упражнения

3.5.1. Реализуйте классы SET и HashSET как классы-оболочки, выполненные в виде клиентов классов ST и HashST соответственно (используйте фиктивные значения и игнорируйте их).

3.5.2. Разработайте реализацию SequentialSearchSET, взяв код SequentialSearchST и убрав из него все, что связано со значениями.

3.5.3. Разработайте реализацию BinarySearchSET, взяв код BinarySearchST и убрав из него все, что связано со значениями.

3.5.4. Разработайте классы HashSTint и HashSTdouble для поддержки множеств ключей для примитивных типов int и double соответственно. (Преобразуйте в коде LinearProbingHashST обобщенные типы в примитивные.)

3.5.5. Разработайте классы STint и STdouble для поддержки упорядоченных таблиц имен, в которых ключи являются примитивными типами int и double соответственно. (Преобразуйте в коде RedBlackBST обобщенные типы в примитивные.) Проверьте полученное решение с вариантом SparseVector в качестве клиента.

3.5.6. Разработайте классы HashSETint и HashSETdouble для поддержки множеств ключей для примитивных типов int и double соответственно. (Уберите код работы со значениями из решения упражнения 3.5.4.)

3.5.7. Разработайте классы SETint и SETdouble для поддержки упорядоченных множеств ключей для примитивных типов int и double соответственно. (Уберите код работы со значениями из решения упражнения 3.5.5.)

3.5.8. Измените код LinearProbingHashST, чтобы иметь возможность хранить в таблице повторяющиеся ключи. Метод get() должен возвращать *любое* значение, связанное с указанным ключом, а метод delete() должен удалять из таблицы *все* элементы с ключами, равными указанному.

3.5.9. Измените код BST, чтобы иметь возможность хранить в дереве повторяющиеся ключи. Метод get() должен возвращать *любое* значение, связанное с указанным ключом, а метод delete() должен удалять из таблицы *все* элементы с ключами, равными указанному.

3.5.10. Измените код RedBlackBST, чтобы иметь возможность хранить в дереве повторяющиеся ключи. Метод get() должен возвращать *любое* значение, связанное с указанным ключом, а метод delete() должен удалять из таблицы *все* элементы с ключами, равными указанному.

3.5.11. Разработайте класс MultiSET, который похож на SET, но допускает одинаковые ключи и, таким образом, реализует математическое понятие *мультимножества.*

3.5.12. Измените код LookupCSV так, чтобы связать с каждым ключом все значения, соответствующие этому ключу, во входных данных (а не только самое последнее, как в абстракции ассоциативного массива).

3.5.13. Преобразуйте код LookupCSV в программу RangeLookupCSV, которая принимает из стандартного ввода два граничных значения ключей и выводит в .csv-файл все такие пары ключ-значение, ключи которых находятся в указанном диапазоне.

3.5.14. Разработайте и проверьте статический метод `invert()`, который принимает в качестве аргумента объект `ST<String, Bag<String>>` и возвращает таблицу имен того же типа с инвертированной исходной таблицей имен.

3.5.15. Напишите программу, которая принимает из стандартного ввода строку, а из командной строки целое число k и выводит в стандартный вывод упорядоченный список k-грамм, найденных в строке, вместе с их индексами в исходной строке.

3.5.16. Добавьте в класс `SparseVector` метод `sum()`, который принимает в качестве аргумента `SparseVector` и возвращает объект `SparseVector`, равный поэлементной сумме данного вектора и вектора аргумента. *Примечание*: нужно, чтобы метод `delete()` учитывал особый случай, когда элемент равен 0 (с учетом точности).

Творческие задачи

3.5.17. *Математические множества.* Разработайте реализацию API `MathSET` для работы с (изменяемыми) математическими множествами (рис. 3.5.7). Используйте таблицу имен. *Дополнительно*: представляйте множества массивами значений `boolean`.

public class **MathSET<Key>**

	`MathSET(Key[] universe)`	*создание множества*
`void`	`add(Key key)`	*добавление ключа* `key` *в множество*
`MathSET<Key>`	`complement()`	*множество всех возможных ключей, которые отсутствуют в данном множестве*
`void`	`union(MathSET<Key> a)`	*занесение в множество всех ключей из* `a`*, которые отсутствуют в данном множестве*
`void`	`intersection(MathSET<Key> a)`	*удаление из данного множества всех ключей, отсутствующих в* **a**
`void`	`delete(Key key)`	*удаление ключа* `key` *из множества*
`boolean`	`contains(Key key)`	*находится ли* `key` *в множестве?*
`boolean`	`isEmpty()`	*пусто ли множество?*
`int`	`size()`	*количество ключей в множестве*

Рис. 3.5.7. API для типа данных простого множества

3.5.18. *Мультимножества.* После выполнения упражнений 3.5.2 и 3.5.3 и предыдущего упражнения разработайте API `MultiHashSET` и `MultiSET` для мультимножеств (множества, которые могут содержать одинаковые ключи) и упорядоченных мультимножеств соответственно.

3.5.19. *Одинаковые ключи в таблицах имен.* Пусть API `MultiST` (упорядоченное и неупорядоченное множество) совпадают с API таблицы имен, представленными на рис. 3.1.1 и 3.1.2, но теперь в них разрешены одинаковые ключи. Метод `get()` должен возвращать *любое* значение, связанное с указанным ключом, и нужно еще добавить метод

```
Iterable<Value> getAll(Key key)
```

который возвращает *все* значения, связанные с указанным ключом. Взяв за основу код классов SeparateChainingST и BinarySearchST, разработайте реализации SeparateChainingMultiST и BinarySearchMultiST для этих API.

3.5.20. *Алфавитный указатель (конкордация).* Напишите клиент Concordance класса ST, который выводит в стандартный вывод алфавитный указатель строк из потока стандартного ввода (см. подраздел “Инвертированный индекс”).

3.5.21. *Инвертированный алфавитный указатель.* Напишите программу InvertedConcordance, которая принимает из стандартного ввода алфавитный указатель и выводит исходную строку в поток стандартного вывода. *Примечание*: это вычисление связывают с известной историей по расшифровке “списков Мертвого моря”. Команда, обнаружившая исходные записи, ввела правило секретности, в результате которого был опубликован только алфавитный указатель. Спустя некоторое время другие исследователи сообразили, как инвертировать этот указатель, и таким образом был опубликован полный текст.

3.5.22. *Полностью индексированный CSV.* Реализуйте клиент FullLookupCSV класса ST, который создает массив объектов ST (по одному для каждого поля), и клиент тестирования, который позволяет указывать поля ключа и значения в каждом запросе.

3.5.23. *Разреженные матрицы.* Разработайте API и реализацию для разреженных двумерных матриц. Напишите методы матричного сложения и умножения и включите конструкторы для векторов-строк и векторов-столбцов.

3.5.24. *Поиск в неперекрывающихся интервалах.* Пусть имеется список неперекрывающихся интервалов элементов. Напишите функцию, которая принимает в качестве аргумента элемент и определяет, в каком интервале (если такой есть) находится данный элемент. Например, если элементами являются целые числа и заданы интервалы 1643-2033, 5532-7643, 8999-10332 и 5666653-5669321, то элемент 9122 находится в третьем интервале, а 8122 — ни в каком из них.

3.5.25. *Задача секретаря.* Недавно в одном крупном северо-восточном университете секретарь отправил одного преподавателя одновременно проводить занятия в двух различных группах. Помогите секретарю избежать таких ошибок в будущем и опишите метод для обнаружения подобных конфликтов. Для простоты примите, что все занятия длятся по 50 минут и начинаются в 9:00, 10:00, 11:00, 13:00, 14:00 и 15:00.

3.5.26. *Кеш LRU.* Напишите структуру данных, которая поддерживает операции доступа и удаления. Операция доступа заносит элемент в структуру данных, если его там нет. Операция удаления удаляет и возвращает элемент, к которому обращение произошло раньше всех остальных (least recently used — LRU). *Совет*: храните элементы в порядке обращений в дважды связном списке, и отдельно храните указатели на первый и последний узлы. Используйте таблицу имен, где ключи = элементы, а значения = местоположения. При доступе к элементу удалите его из связного списка и снова вставьте в начало. При удалении элемента уничтожьте его в конце и удалите из таблицы имен.

3.5.27. *Список.* Разработайте реализацию API, приведенного на рис. 3.5.8. *Совет*: используйте две таблицы имен: одну для эффективного поиска i-го элемента в

списке, а другую — для эффективного поиска по элементу. (Java-интерфейс `java.util.List` содержит подобные методы, но не предоставляет реализацию, которая бы эффективно поддерживала все операции.)

`public class **List**<Item> implements Iterable<Item>`

	`List()`	*создание списка*
`void`	`addFront(Item item)`	*занесение элемента* `item` *в начало*
`void`	`addBack(Item item)`	*занесение элемента* `item` *в конец*
`Item`	`deleteFront()`	*удаление из начала*
`Item`	`deleteBack()`	*удаление из конца*
`void`	`delete(Item item)`	*удаление элемента* `item` *из списка*
`void`	`add(int i, Item item)`	*занесение элемента* `item` *в* `i`*-ю позицию списка*
`void`	`delete(int i)`	*удаление* `i`*-го элемента из списка*
`boolean`	`contains(Key key)`	*находится ли* `key` *в списке?*
`boolean`	`isEmpty()`	*пуст ли список?*
`int`	`size()`	*количество ключей в списке*

Рис. 3.5.8. *API для типа данных списка*

3.5.28. *Одноочередь.* Создайте тип данных — очередь, в которую любой элемент можно вставить только один раз. Для отслеживания всех когда-либо вставленных элементов задействуйте таблицу существования имен, и игнорируйте запросы на повторную вставку таких элементов.

3.5.29. *Таблица имен со случайным доступом.* Создайте тип данных, который поддерживает вставку пары ключ-значение, поиск ключа с возвратом связанного с ним значения и удаление случайного ключа с его возвратом. *Совет*: используйте сочетание таблицы имен и рандомизированной очереди.

Эксперименты

3.5.30. *Дубликаты (еще раз).* Выполните упражнение 2.5.31, воспользовавшись фильтром `Dedup` из листинга 3.5.1. Сравните значения времени выполнения для обоих способов. Затем проведите эксперименты с `Dedup` для $N = 10^7$, 10^8 и 10^9, повторите эксперименты для случайных значений `long` и проанализируйте полученные результаты.

3.5.31. *Проверка грамматики.* Клиент `BlackFilter` (см. листинг 3.5.2) использует в качестве аргумента командной строки файл `dictionary.txt` с сайта книги и выводит все слова с ошибками, которые имеются в текстовом файле, вводимом из стандартного ввода. Сравните с помощью этого клиента производительность классов `RedBlackBST`, `SeparateChainingHashST` и `LinearProbingHashST` для файла `WarAndPeace.txt` (доступен на сайте книги); проанализируйте полученные результаты.

3.5.32. *Словарь.* Рассмотрите поведение клиента вроде LookupCSV в ситуации, где производительность важна. Для этого имитируйте ситуацию с генерацией запросов, а не с чтением команд из стандартного ввода, и выполните тесты производительности для больших входных данных и большого количества запросов.

3.5.33. *Индексация.* Рассмотрите поведение клиента вроде LookupIndex в ситуации, где производительность важна. Для этого имитируйте ситуацию с генерацией запросов, а не с чтением команд из стандартного ввода, и выполните тесты производительности для больших входных данных и большого количества запросов.

3.5.34. *Разреженный вектор.* Экспериментально сравните производительность умножения матрицы на вектор с помощью класса SparseVector со стандартной реализацией на основе массивов.

3.5.35. *Примитивные типы.* Оцените удобство использования примитивных типов, соответствующих типам Integer и Double, для классов LinearProbingHashST и RedBlackBST. Какова экономия памяти и времени для большого количества поисков в больших таблицах?

Глава 4

Графы

Попарные соединения между элементами играют важную роль в огромном множестве вычислительных приложений. Отношения, обозначаемые такими соединениями, сразу же порождают массу вполне естественных вопросов. Существует ли путь от одного элемента к другому по соединениям? Сколько других элементов соединено с указанным элементом? Какова кратчайшая цепочка соединений между двумя указанными элементами?

Для моделирования таких ситуаций мы используем абстрактную математическую модель — *графы*. В данной главе мы подробно изучим основные свойства графов, которые необходимы для понимания различных алгоритмов, позволяющих получить ответы на вопросы вроде приведенных выше. Эти алгоритмы служат отправной точкой для решения задач в важных приложениях, решение которых невозможно было бы получить без хорошей алгоритмической технологии.

Теория графов — крупный раздел математики, который интенсивно развивается уже сотни лет. За это время были открыты многие важные и полезные свойства графов, разработаны многие важные алгоритмы, но множество сложных задач еще ждет своего изучения и решения. В этой главе мы ознакомимся с несколькими фундаментальными алгоритмами на графах, которые применяются в самых различных приложениях.

Как и многие другие рассмотренные нами категории задач, алгоритмическое исследование графов началось относительно недавно. Возраст нескольких фундаментальных алгоритмов измеряется столетиями, но большинство интересных решений найдено лишь в последние несколько десятилетий — и они возникли благодаря развитию алгоритмической технологии, к изучению которой мы сейчас приступаем. Даже простейшие алгоритмы на графах приводят к полезным компьютерным программам, а те нетривиальные алгоритмы, с которыми мы познакомимся, являются наиболее элегантными и полезными из всех известных. Для демонстрации распространенности приложений, в которых применяется обработка графов, мы начнем наше знакомство в этой плодородной области с рассмотрения нескольких примеров (табл. 4.0.1).

- *Карты*. Человека, планирующего поездку, могут заинтересовать вопросы вроде "Как быстрее добраться от Провиденса до Принстона?" Для ответа на такие вопросы нужно обрабатывать информацию о соединениях (дорогах) между элементами (перекрестками).
- *Веб-контент*. При просмотре Интернет-сайтов мы попадаем на страницы, содержащие ссылки на другие страницы, щелкая на которых, можно переходить со страницы на страницу. Вся мировая Сеть представляет собой граф, где элементы являются страницами, а соединения — ссылками. Алгоритмы обработки графов являются важнейшими компонентами поисковых механизмов, которые помогают найти информацию в Сети.
- *Микросхемы*. Кристалл микросхемы содержит такие устройства, как транзисторы, резисторы и конденсаторы, соединенные между собой самым замысловатым образом. Для управления машинами, которые создают микросхемы и проверяют их работоспособность, используются компьютеры. При этом необходимо отвечать как на простые вопросы вроде "Имеется ли замыкание в сети?", так и на сложные вопросы наподобие "Можно ли расположить эту схему на кристалле без пересечения проводников?". Ответ на первый вопрос зависит только от свойств соединений (проводников), а для второго вопроса нужна подробная информация о проводниках, устройствах, которые соединяются этими проводниками, и физических ограничениях, присущих чипу.

- *Расписания.* Процесс производства требует выполнения различных работ с учетом набора ограничений, которые указывают, что некоторые работы невозможно начать, если не будут завершены какие-то другие работы. Как можно составить график выполнения этих работ, чтобы не нарушать ограничения и завершить весь процесс за минимальное время?
- *Финансы.* Розничные торговцы и финансовые учреждения отслеживают заказы на покупку/продажу на рынке. Соединение в данном случае представляет собой перевод денег или пересылку товаров между учреждениями и клиентами. Знание структуры этих соединений может помочь пониманию природы всего рынка.
- *Сопоставление.* Студенты принимают участие в работе таких учреждений, как социальные клубы, университеты или медицинские училища. Элементы соответствуют студентам и учреждениям, а соединения соответствуют членству. Необходимо найти способы сопоставления заинтересованных студентов и доступных вакансий.
- *Компьютерные сети.* Компьютерная сеть состоит из взаимосвязанных вычислительных площадок, которые посылают, перенаправляют и принимают сообщения различных типов. Необходимо знание о природе структуры этих взаимосвязей, позволяющее так провести кабели и разместить концентраторы, чтобы эффективно обрабатывать трафик.
- *Программное обеспечение.* Компилятор строит графы для представления взаимосвязей между модулями в большой программной системе. Элементами являются различные классы или модули, составляющие систему, а соединения представляют собой либо возможность метода из одного класса вызывать метод из другого класса (статический анализ), либо реальные вызовы при работе системы (динамический анализ). Необходим анализ графа для определения, как эффективнее выделять ресурсы такой программе.
- *Социальные сети.* При участии в социальной сети создаются явные связи с вашими друзьями. Элементы соответствуют людям, а соединения — друзьям или последователям. Понимание свойств таких сетей — пример современного применения обработки графов, которые интенсивно изучаются не только компаниями, поддерживающими эти сети, но и исследователями в политике, дипломатии, индустрии досуга, образовании, маркетинге и многих других областях.

Таблица 4.0.1. Типичные применения графов

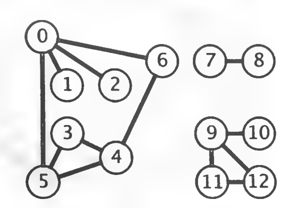
Применение	Элемент	Соединение
Карта	Перекресток	Дорога
Веб-контент	Страница	Ссылка
Микросхема	Устройство	Проводник
Расписание	Работа	Ограничение
Финансы	Клиент	Транзакция
Сопоставление	Студент	Членство
Компьютерная сеть	Сайт	Соединение
Программное обеспечение	Метод	Вызов
Социальная сеть	Человек	Дружба

Эти примеры демонстрируют распространенность приложений, в которых графы являются удобной абстракцией, а также диапазон вычислительных задач, которые могут встретиться при работе с графами. Изучены уже тысячи таких задач, но многие из них можно решать в контексте одной из нескольких базовых моделей графов — и в этой главе мы изучим наиболее важные из них. В практических приложениях объем используемых данных часто бывает просто огромным, поэтому эффективные алгоритмы означают разницу между возможностью и невозможностью решения этих задач.

Чтобы помочь вам оценить масштаб применения графов, мы рассмотрим четыре наиболее важных модели графов: *неориентированные графы* (с простыми соединениями), *орграфы* (в которых важно направление каждого соединения), *графы с взвешенными ребрами* (в которых с каждым соединением связан некоторый вес) и *орграфы с взвешенными ребрами* (в которых каждое соединение имеет и направление, и вес).

4.1. Неориентированные графы

Вначале мы будем рассматривать модели графов, в которых *ребра* представляют собой просто соединения между *вершинами*. Там, где потребуется отличать эту модель от других моделей (например, в заголовке данного раздела), мы будем использовать термин *неориентированный граф*, но, поскольку это простейшая модель, мы начнем со следующего определения:



Определение. *Граф* — это множество *вершин* и коллекция *ребер*, каждое из которых соединяет пару вершин.

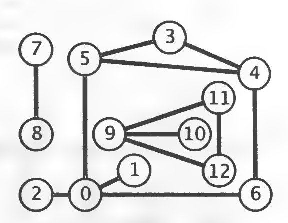


Рис. 4.1.1. Два изображения одного и того же графа

Имена вершин в определении не важны, но нам все-таки нужен способ называть их. По соглашению, мы будем использовать в графе с V вершинами имена для вершин от 0 до $V-1$. Основная причина выбора этой системы — она облегчает написание кода, который эффективно обращается к информации, соответствующей каждой вершине, с помощью индексов массива. Нетрудно задействовать таблицу имен и установить взаимно однозначное соответствие V произвольных имен вершин с V целыми числами от 0 до $V-1$ (см. раздел "Символьные графы"), поэтому удобство использования индексов в качестве имен вершин не приводит к потере общности (и почти не приводит к ухудшению эффективности). Мы будет использовать обозначение v-w для ребра, соединяющего вершины v и w; то же самое ребро можно обозначить и как w-v.

На рисунках вершины графов обозначаются кружочками, а ребра — соединяющими их линиями. Такие чертежи дают наглядное представление о структуре графа, но это представление не всегда верно, т.к. определение графа не зависит от его изображения. Например, два чертежа на рис. 4.1.1 представляют собой один и тот же граф, т.к. граф — это просто (неупорядоченное) множество его вершин и (неупорядоченная) коллекция его ребер (пар вершин).

Аномалии

Наше определение допускает существование двух простых аномалий (рис. 4.1.2).

- *Петля* — ребро, соединяющее некоторую вершину с самой собой.
- Два ребра, соединяющие одну и ту же пару вершин, называются *параллельными*.

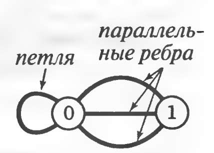


Рис. 4.1.2. Аномалии в графе

Математики иногда называют графы с параллельными ребрам *мультиграфами*, а графы без параллельных ребер и циклов — *простыми* графами. Обычно наши реализации будут допускать петли и параллельные ребра (т.к. они возникают в приложениях), но мы не будем включать их в примеры. Поэтому для названия ребра достаточно будет указать две вершины, которые оно соединяет.

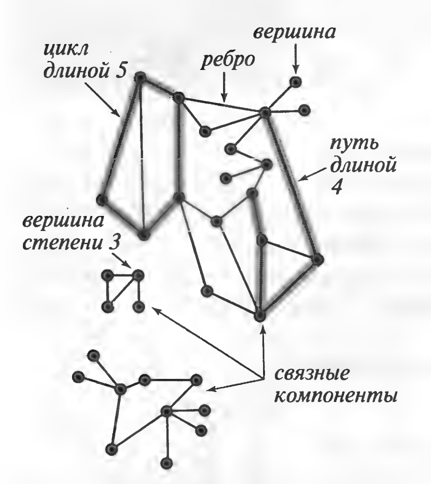


Рис. 4.1.3. Внутренний состав графа

Термины

С графами связан значительный объем терминологии (рис. 4.1.3). Большинство из этих терминов имеют вполне понятные определения, и, чтобы не распыляться, мы приведем их в одном месте, здесь.

Если имеется ребро, соединяющее две вершины, мы говорим, что эти вершины *смежные*, а ребро *инцидентно* обеим вершинам. *Степень* вершины — это количество ребер, инцидентных с ней. *Подграф* — это подмножество ребер графа (и связанные с ними вершины), которое составляет граф. Во многих вычислительных задачах используются различные подграфы; особенно интересны ребра, которые проводят через *последовательность* вершин графа.

Определение. *Путь* в графе — это последовательность вершин, соединенных ребрами. *Простой путь* — это путь без повторяющихся вершин. *Цикл* — это путь, содержащий хотя бы одно ребро, в котором первая и последняя вершины совпадают. *Простой цикл* — это цикл без повторяющихся ребер и вершин (за исключением обязательного совпадения первой и последней вершин). *Длина* пути или цикла равна количеству его ребер.

Обычно мы будем работать с простыми циклами и простыми путями и поэтому будем опускать описатель *простой*, а когда нам понадобится разрешить повторение вершин, мы будем говорить о путях и циклах *общего вида*. Мы говорим, что одна вершина *связана* (соединена) с другой вершиной, если существует путь, содержащий обе эти вершины. Мы будем использовать обозначение вроде `u-v-w-x` для пути из вершины `u` в вершину `x` и обозначение `u-v-w-x-u` для цикла из вершины `u` через вершины `v`, `w` и `x` снова в `u`. Мы познакомимся с несколькими алгоритмами, которые находят пути и циклы. Более того, пути и циклы помогают рассматривать структурные свойства графа как целого.

Определение. Граф называется *связным*, если в нем существует путь из каждой вершины в каждую другую вершину. *Не связный* граф состоит из множества *связных компонентов*, которые представляют собой максимальные связные подграфы.

Если представить вершины физическими объектами вроде узлов или бусинок, а ребра — физическими связями наподобие ниток или проволок, то связный граф останется целым, если потянуть за любую из его вершин, а не связный граф распадется на две или более частей. Обычно обработка графов сводится к поочередной обработке его связных компонентов.

Ациклический граф — это граф без циклов. Несколько алгоритмов, с которыми нам предстоит познакомиться, выполняют в заданном графе поиск ациклических подграфов, которые удовлетворяют определенным условиям. Для работы с такими структурами нам понадобится дополнительная терминология.

Определение. *Дерево* — это ациклический связный граф. Множество не связанных между собой деревьев называется *лесом*. *Остовное дерево* связного графа — это подграф, который содержит все вершины исходного графа и является единым деревом. *Остовный лес* графа — это совокупность остовных деревьев связных компонентов этого графа (рис. 4.1.4 и 4.1.5).



Рис. 4.1.4. Дерево

Данное определение дерева довольно общее: с некоторыми уточнениями оно охватывает и деревья, которые мы обычно используем для описания поведения программы (иерархия вызовов функций), и структуры данных (ДБП, 2-3-деревья и т.д.). Математические свойства деревьев хорошо изучены и интуитивно понятны, поэтому мы будем формулировать их без доказательств. Например, граф G с V вершинами является деревом в том и только в том случае, если он удовлетворяет одному из следующих пяти условий:

- G содержит $V-1$ ребер и не содержит циклов;
- G содержит $V-1$ ребер и связный;
- G связный, но при удалении любого ребра перестает быть связным;
- G ацикличный, но добавление любого ребра создает в нем цикл;
- каждую пару вершин в G соединяет в точности один простой путь.

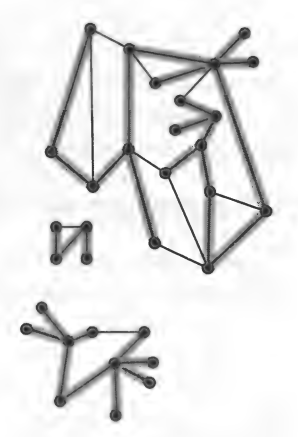


Рис. 4.1.5. Остовный лес

Несколько алгоритмов, которые мы рассмотрим в данной главе, находят остовные деревья и леса, и эти свойства играют важную роль в их анализе и реализации.

Плотность графа — это отношение количества ребер к количеству всех возможных пар вершин, которые можно соединить ребрами. *Разреженный* граф содержит относительно немного из числа возможных ребер, а *насыщенный* граф содержит относительно немного незадействованных ребер (рис. 4.1.6). Обычно граф считается разреженным, если количество его различных ребер лишь в небольшое число раз превышает V, иначе он насыщенный.

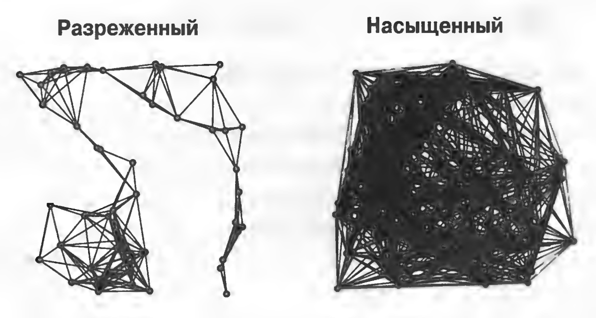


Рис. 4.1.6. Два графа ($V = 50$)

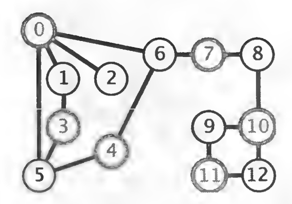


Рис. 4.1.7. *Двудольный граф*

Это грубое правило оставляет смутно определенным некоторое множество графов (с количеством ребер, скажем, $\sim c V^{3/2}$), однако в приложениях различие между разреженными и насыщенными графами обычно видно четко. Приложения, с которыми мы будем иметь дело, почти всегда работают с разреженными графами.

Двудольный граф — это граф, вершины которого можно разбить на два таких множества, что любое ребро графа соединяет вершину из одного множества с вершиной из другого множества. На рис. 4.1.7 приведен пример двудольного графа, где одно множество вершин показано серым цветом, а другое множество — черным. Двудольные графы естественным образом возникают во многих ситуациях, одну из которых мы подробно рассмотрим в конце данного раздела.

Теперь мы готовы перейти к изучению алгоритмов обработки графов. Сначала мы введем API и его реализацию для типа данных графа, а затем рассмотрим классические алгоритмы поиска на графах и определения связных компонентов. В конце раздела будут описаны реальные приложения, в которых имена вершин могут не быть целыми числами, а графы могут содержать огромные количества вершин и ребер.

Тип данных неориентированного графа

Разработку алгоритмов обработки графов мы начнем с API, который определяет фундаментальные операции на графах (рис. 4.1.8). Эта схема позволяет работать с задачами обработки графов в диапазоне от элементарных операций сопровождения до мудреных решений сложных задач.

public class **Graph**		
	`Graph(int V)`	*создание графа с* `V` *вершинами без ребер*
	`Graph(In in)`	*чтение графа из входного потока* `in`
`int`	`V()`	*количество вершин в графе*
`int`	`E()`	*количество ребер в графе*
`void`	`addEdge(int v, int w)`	*добавление в граф ребра* `v-w`
`Iterable<Integer>`	`adj(int v)`	*вершины, смежные с* `v`
`String`	`toString()`	*строковое представление*

Рис. 4.1.8. *API для неориентированного графа*

Этот API содержит два конструктора, методы для возврата количества вершин и ребер, метод для добавления ребра, метод `toString()` и метод `adj()`, который позволяет клиентскому коду перебирать вершины, смежные с указанной вершиной (порядок перебора не определен). Интересно, что все алгоритмы, рассматриваемые в данном разделе, можно построить на этой базовой абстракции, встроенной в `adj()`.

Второй конструктор предполагает, что входные данные содержат $2E + 2$ целочисленных значений: V, потом E, а затем E пар значений от 0 до $V-1$, где каждая пара означает ребро. В качестве примера на рис. 4.1.9 приведены два графа — `tinyG.txt` и `mediumG.txt`.

Несколько примеров клиентского кода для класса `Graph` приведено в табл. 4.1.1.

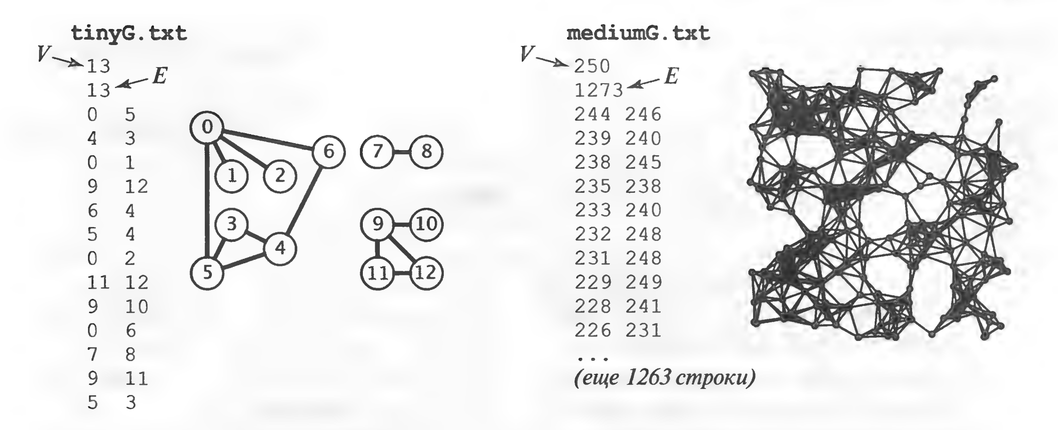


Рис. 4.1.9. Формат входных данных для конструктора класса `Graph` *(два примера)*

Таблица 4.1.1. Типичный код обработки графов

Задача	Реализация
Вычисление степени вершины v	см. код ниже

Вычисление степени вершины v

```
public static int degree(Graph G, int v)
{
   int degree = 0;
   for (int w : G.adj(v)) degree++;
   return degree;
}
```

Нахождение максимальной степени

```
public static int maxDegree(Graph G)
{
   int max = 0;
   for (int v = 0; v < G.V(); v++)
      if (degree(G, v) > max)
         max = degree(G, v);
   return max;
}
```

Вычисление средней степени

```
public static int avgDegree(Graph G)
{  return 2 * G.E() / G.V();  }
```

Подсчет петель

```
public static int numberOfSelfLoops(Graph G)
{
   int count = 0;
   for (int v = 0; v < G.V(); v++)
      for (int w : G.adj(v))
         if (v == w) count++;
   return count/2;   // каждое ребро посчитано дважды
}
```

Текстовое представление списков смежности графа (метод экземпляров в Graph)

```
public String toString()
{
   String s = V + " вершин, " + E + " ребер\n";
   for (int v = 0; v < V; v++)
   {
      s += v + ": ";
      for (int w : this.adj(v))
         s += w + " ";
      s += "\n";
   }
   return s;
}
```

Возможные представления

Следующее важное решение, которое необходимо принимать в обработке графов — представление графа (структура данных), которое будет применяться для реализации API. Имеются два базовых требования.

- Необходима *память* для хранения типов графов, которые, скорее всего, встретятся нам в приложениях.
- Нужно разработать реализации методов экземпляров `Graph`, эффективные в смысле *времени* выполнения — те базовые методы, которые понадобятся для разработки клиентов обработки графов.

Эти требования не очень четкие, но они все-таки помогают сделать выбор из трех структур данных, которые непосредственно следуют из определения графа.

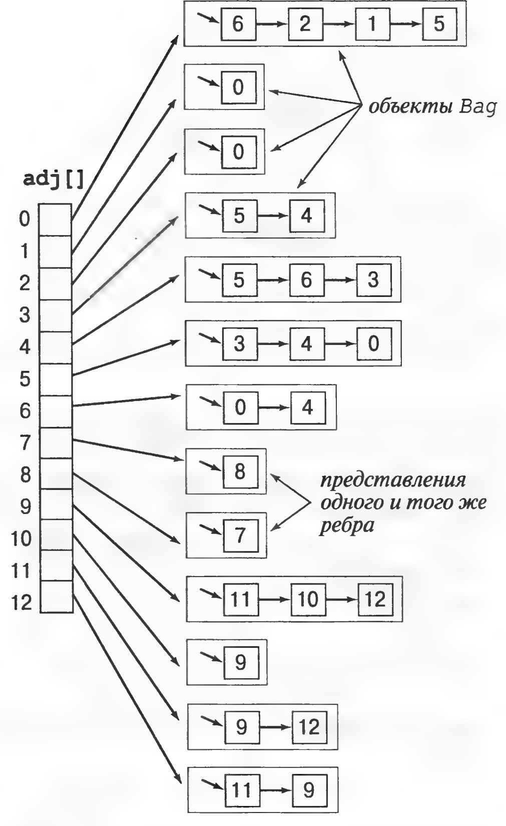


Рис. 4.1.10. *Представление списками смежности (неориентированный граф)*

- *Матрица смежности.* В ней используется логический массив $V \times V$, в котором элемент в строке `v` и столбце `w` равен `true` тогда и только тогда, когда существует ребро между вершинами `v` и `w`, а иначе равен `false`. Это представление не удовлетворяет первому требованию: совсем не редко встречаются графы с миллионами вершин, а память, необходимая для размещения V^2 логических значений — слишком неподъемное требование.
- *Массив ребер.* Используется класс `Edge` с двумя переменными экземпляров типа `int`. Это непосредственное представление простое, но оно также непригодно, по второму требованию: для реализации метода `adj()` приходится просматривать все ребра графа.
- *Массив списков смежности* (рис. 4.1.10). Здесь используется массив списков, индексированных вершинами, и в них содержатся вершины, смежные с каждой вершиной. Эта структура данных удовлетворяет обоим требованиям, и поэтому будет применяться в данной главе.

Кроме этих соображений производительности, при внимательном анализе выявляются и другие критерии, которые могут оказаться важными в некоторых случаях. Например, если разрешить наличие параллельных ребер, то невозможно использовать матрицу смежности, т.к. в ней невозможно их представить.

Структура данных для списков смежности

Стандартное представление не насыщенного графа называется *структурой данных для списков смежности*, где все вершины, смежные с каждой вершиной, хранятся в связном списке, соответствующем этой вершине. Мы используем массив списков, который позволяет по указанной вершине непосредственно перейти к ее списку. Для реализации списков мы воспользуемся АТД `Bag` из раздела 1.3 с реализацией связными списками: так мы сможем добавлять новые ребра за константное время и перебирать смежные вершины за константное время в расчете на одну смежную вершину. Реализация `Graph` в листинге 4.1.1 основана на этом подходе, а на рис. 4.1.11 приведены структуры данных, построенные для файла `tinyG.txt`. Чтобы добавить ребро, соединяющее вершины `v` и `w`, мы добавляем вершину `w` в список смежных вершин `v` и вершину `v` в список смежных вершин `w`. То есть каждое ребро присутствует в структуре данных *дважды*. Приведенная реализация API `Graph` имеет следующие характеристики производительности.

- Объем занимаемой памяти пропорционален $V + E$.
- Добавление ребра за константное время.
- Перебор вершин, смежных с `v`, за время, пропорциональное степени `v` (константное время на обработку каждой смежной вершины).

Листинг 4.1.1. Тип данных графа

```
public class  Graph
{
  private final int V;        // количество вершин
  private int E;              // количество ребер
  private Bag<Integer>[] adj; // списки смежных вершин
  public Graph(int V)
  {
    this.V = V; this.E = 0;
    adj = (Bag<Integer>[]) new Bag[V];       // Создание массива списков.
    for (int v = 0; v < V; v++)              // Вначале все списки
      adj[v] = new Bag<Integer>();           //   создаются пустыми.
  }
  public Graph(In in)
  {
    this(in.readInt());       // Чтение V и создание графа.
    int E = in.readInt();     // Чтение E.
    for (int i = 0; i < E; i++)
    { // Добавление ребра.
      int v = in.readInt();   // Чтение вершины,
      int w = in.readInt();   //   чтение другой вершины,
      addEdge(v, w);          //   и добавление соединяющего их ребра.
    }
  }
  public int V()  {  return V;  }
  public int E()  {  return E;  }
  public void addEdge(int v, int w)
  {
    adj[v].add(w);            // Добавление w в список вершины v.
    adj[w].add(v);            // Добавление v в список вершины w.
    E++;
  }
  public Iterable<Integer> adj(int v)
  {  return adj[v];  }
}
```

Эта реализация Graph использует индексированный вершинами массив списков, содержащих целые числа. Каждое ребро присутствует дважды: если оно соединяет вершины v и w, то w присутствует в списке вершины v, а v присутствует в списке вершины w. Второй конструктор читает граф из входного потока в формате: V, E, а затем список E пар значений int из диапазона от 0 до $V-1$. Реализацию метода toString() см. в табл. 4.1.1.

Такие характеристики оптимальны для заданного набора операций, который достаточен для приложений обработки графов, рассматриваемых в настоящем разделе. Возможно наличие параллельных ребер и петель (проверка не выполняется). *Примечание*: важно понимать, что порядок добавления ребер к графу определяет порядок появления вершин в массиве списков смежных вершин, который создает конструктор Graph. Один и тот же граф может быть представлен многими различными массивами списков смежности. При использовании конструктора, который читает ребра из входного потока, это означает, что входной формат и порядок расположения ребер в файле определяет порядок расположения вершин в построенном массиве списков смежности. Поскольку наши алгоритмы используют метод adj() и обрабатывают все смежные вершины без учета их порядка в списке, эта разница не влияет на их корректность, но ее следует учитывать при отладке или анализе трассировок. Поэтому мы считаем, что у класса Graph есть клиент тестирования, который читает граф из входного потока с именем, передаваемом в качестве аргумента командной строки, а затем выводит его (с помощью реализации toString() из табл. 4.1.1) для демонстрации порядка вершин в списках смежности (рис. 4.1.11) — именно в этом порядке наши алгоритмы будут их обрабатывать (см. упражнение 4.1.7).

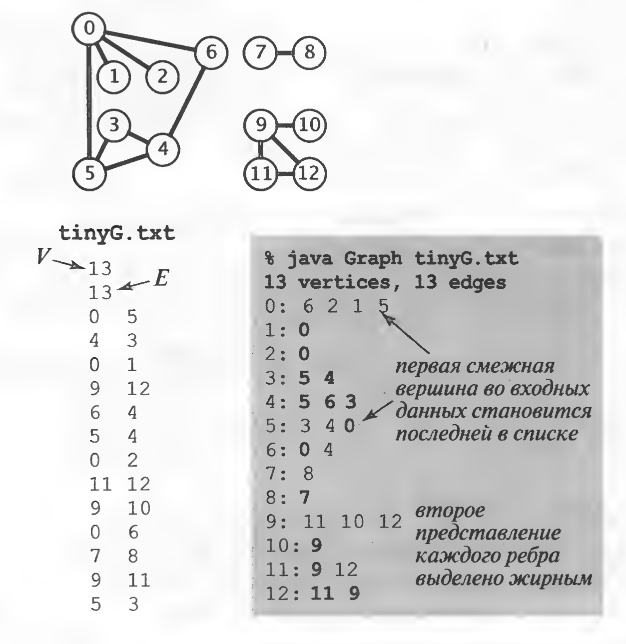


Рис. 4.1.11. Выходные данные для входного списка ребер

Конечно, стоит рассмотреть и другие операции, которые могут оказаться полезными в приложениях — например, методы для следующих операций:

- добавление вершины;
- удаление вершины.

Для поддержки таких операций можно расширить API, чтобы использовать не массив, индексированный вершинами, а таблицу имен — тогда не понадобится и наше соглашение о том, что имена вершин должны быть целочисленными индексами. Можно также рассмотреть следующие методы:

- удаление ребра;
- проверка, содержит ли граф ребро `v-w`.

Для реализации этих методов (и запрета параллельных ребер) можно использовать в связных списках объекты типа не `Bag`, а `SET`. Такой вариант называется представлением *множествами смежности*. В данной книге мы не будем рассматривать его по нескольким причинам.

- В наших клиентах нет необходимости добавлять вершины, удалять вершины и ребра или проверять наличие ребер.
- Если в клиентах такие операции и понадобятся, они обычно выполняются нечасто или для коротких списков смежности, поэтому проще всего использовать примитивную реализацию с перебором списка смежных вершин.
- Представления `SET` и `ST` несколько усложняют код реализации алгоритма и отвлекают внимание от самих алгоритмов.
- В некоторых ситуациях производительность может ухудшиться до $\log V$.

Наши алгоритмы нетрудно адаптировать к иным требованиям (например, запрет параллельных ребер или петель) без излишних потерь производительности. В табл. 4.1.2 приведены характеристики производительности упомянутых здесь вариантов. В типичных приложениях выполняется обработка огромных разреженных графов, поэтому мы будем везде использовать представление списками смежности.

Таблица 4.1.2. Порядок роста трудоемкости для типичных реализаций `Graph`

Базовая структура данных	Память	Добавление ребра `v-w`	Проверка, смежна ли `w` вершине `v`	Перебор вершин, смежных с `v`
Список ребер	E	1	E	E
Матрица смежности	V^2	1	1	V
Списки смежности	$E+V$	1	*степень*(`v`)	*степень*(`v`)
Множества смежности	$E+V$	$\log V$	$\log V$	$\log V+$ *степень*(`v`)

Типичные действия при программировании обработки графов

Поскольку мы собираемся рассмотреть большое количество алгоритмов обработки графов, нам нужно вначале научиться отделять наши реализации от представления графов. Для этого мы для каждой конкретной задачи будем разрабатывать специальный класс, чтобы клиенты могли создавать объекты, необходимые для решения этой задачи. Как правило, при создании структур данных конструктор выполняет некоторую предварительную обработку, которая позволяет более эффективно реагировать на запросы

клиентов. Типичная клиентская программа строит граф, передает этот граф в класс реализации алгоритма (как аргумент конструктора), а затем вызывает методы клиентских запросов, которые позволяют изучить различные свойства графа. Для начала рассмотрим API, приведенный на рис. 4.1.12.

public class **Search**		
	Search(Graph G, int s)	*поиск вершин, связанных с вершиной* s
boolean	marked(int v)	*связана ли* v *с вершиной* s?
int	count()	*сколько вершин связано с* s?

Рис. 4.1.12. *API обработки графов (первоначальный)*

Чтобы отличить вершину, передаваемую в качестве аргумента конструктору, от других вершин графа, мы будем называть ее *исходной* или *источником*. В рассматриваемом API конструктор должен найти в графе вершины, связанные с исходной. Потом клиент вызывает методы экземпляров marked() и count(), чтобы узнать некоторые характеристики графа. Имя marked (помеченный) соответствует способу действия базовых алгоритмов, которые мы будем рассматривать во всей главе: они проходят по путям от источника к другим вершинам в графе и помечают все встреченные вершины. Демонстрационный клиент TestSearch, приведенный в листинге 4.1.2, принимает из командной строки имя входного потока и номер исходной вершины, а затем выводит вершины, связанные с исходной в данном графе, с помощью метода marked() (рис. 4.1.13). Он также вызывает метод count() и сообщает, является ли граф связным (граф связен тогда и только тогда, когда поиск помечает все его вершины).

Листинг 4.1.2. Пример клиента обработки графов (первоначальный API)

```
public class TestSearch
{
   public static void main(String[] args)
   {
      Graph G = new Graph(new In(args[0]));
      int s = Integer.parseInt(args[1]);
      Search search = new Search(G, s);
      for (int v = 0; v < G.V(); v++)
         if (search.marked(v))
            StdOut.print(v + " ");
      StdOut.println();
      if (search.count() != G.V())
         StdOut.print("НЕ ");
      StdOut.println("связный");
   }
}
```

```
% java TestSearch tinyG.txt 0
0 1 2 3 4 5 6
НЕ связный
% java TestSearch tinyG.txt 9
9 10 11 12
НЕ связный
```

Мы уже знакомы с одним способом реализации API `Search` — это алгоритмы объединения-поиска из главы 1. Конструктор может построить объект `UF`, выполнить операцию `union()` для каждого из ребер графа и реализовать вызов `marked(v)` с помощью вызова `connected(s, v)`. Для реализации `count()` понадобится взвешенная реализация `UF` и расширение API, чтобы использовать метод `count()`, возвращающий `wt[find(v)]` (см. упражнение 4.1.8). Такая реализация вполне проста и эффективна, но реализация, которую мы рассмотрим ниже, еще проще и эффективнее. Она основана на *поиске в глубину* — фундаментальном рекурсивном методе, который проходит по ребрам графа и отыскивает вершины, связанные с исходной. На поиске в глубину основано несколько алгоритмов обработки графов, которые будут рассмотрены в данной главе.

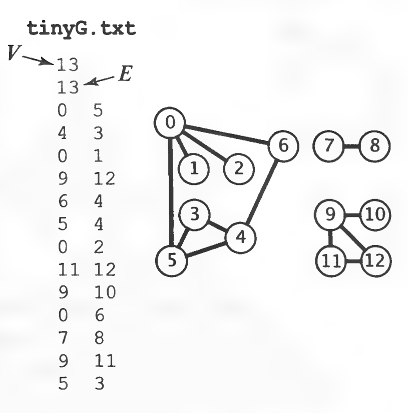


Рис. 4.1.13. *Демонстрационный пример графа*

Поиск в глубину

Свойства графа часто исследуются с помощью систематического просмотра каждой его вершины и каждого ребра. Определение некоторых простых свойств графов — например, подсчет степеней всех его вершин — нетрудно выполнить с помощью обычного просмотра каждой вершины (в произвольном порядке). Но многие другие свойства графов связаны с путями, поэтому естественно исследовать их с помощью перемещения от вершины к вершине по ребрам графа. Почти все алгоритмы обработки графов, которые мы будем изучать, используют как раз такую базовую абстрактную модель, хотя и с различными стратегиями. Простейшей из них является классический метод, который мы сейчас рассмотрим.

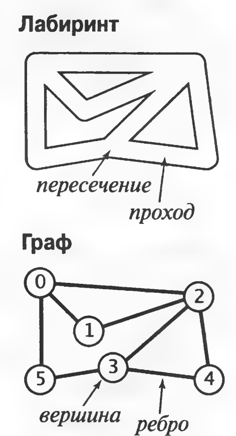


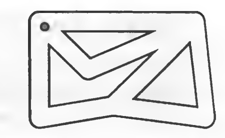
Рис. 4.1.14. *Эквивалентные модели лабиринта*

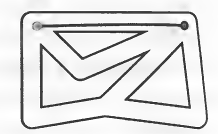
Поиск в лабиринте

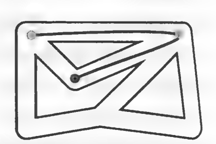
Процесс поиска в графе удобно рассматривать как эквивалентную задачу с длинной и интересной историей — поиск пути в лабиринте, который состоит из проходов и пересечений этих проходов. Некоторые лабиринты можно пройти, пользуясь простым правилом, однако для большинства лабиринтов требуется более сложная стратегия. Использование терминов *лабиринт* вместо *граф*, *проход* вместо *ребро* и *пересечение* вместо *вершина* — это просто семантическое различие, но оно позволяет лучше почувствовать задачу (рис. 4.1.14). Один из способов пройти лабиринт и не заблудиться в нем известен еще с античных времен (как минимум со времени возникновения легенды о Тесее и Минотавре), он называется *правилом Тремо* (Tremaux exploration). Чтобы пройти все проходы лабиринта, нужно (рис. 4.1.15):

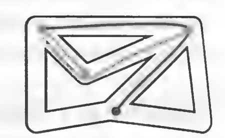
- пойти в любой непомеченный проход, разматывая за собой нить;
- помечать все пересечения и проходы при первом их посещении;
- возвращаться назад (используя нить) при попадании в помеченное пересечение;
- возвращаться назад, если не осталось непосещенных проходов в пересечениях при отходе назад.

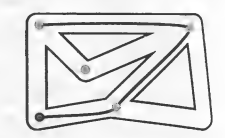
Нить гарантирует, что вы всегда сможете найти обратный путь, а пометки гарантируют, что вы не будете проходить дважды ни один проход или пересечение. Чтобы узнать, что вы просмотрели весь лабиринт, требуется более сложное рассуждение, которое лучше проводить в контексте графов и поиска в них. Правило Тремо — наглядная отправная точка, однако оно кое-чем отличается от прохода по графу, так что теперь мы перейдем к поиску на графах.











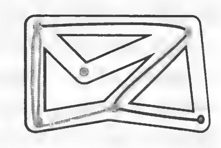


Рис. 4.1.15. Обход по правилу Тремо

Разминка

Классический рекурсивный метод для поиска в связном графе (посещение всех его вершин и ребер) имитирует обход по правилу Тремо, но описать его даже проще. Для поиска на графе нужно вызвать рекурсивный метод посещения вершин. Чтобы посетить вершину, нужно:

- пометить ее как посещенную;
- посетить (рекурсивно) все еще не помеченные вершины, смежные с ней.

Этот метод называется *поиском в глубину* (ПвГ). Основанная на нем реализация нашего API `Search` приведена в листинге 4.1.3. В нем используется массив значений `boolean` для пометки всех вершин, которые связаны с исходной вершиной. Рекурсивный метод помечает указанную вершину и вызывает себя для всех непомеченных вершин из ее списка смежности. Если граф связен, то будут просмотрены все элементы списков смежности.

Листинг 4.1.3. Поиск в глубину

```
public class DepthFirstSearch
{
   private boolean[] marked;
   private int count;

   public DepthFirstSearch(Graph G, int s)
   {
      marked = new boolean[G.V()];
      dfs(G, s);
   }

   private void dfs(Graph G, int v)
   {
      marked[v] = true;
      count++;
      for (int w : G.adj(v))
         if (!marked[w]) dfs(G, w);
   }

   public boolean marked(int w)
   {  return marked[w];  }

   public int count()
   {  return count;  }
}
```

Утверждение А. Поиск в глубину помечает все вершины, связанные с указанной исходной вершиной, за время, пропорциональное сумме их степеней.

Доказательство. Вначале мы докажем, что алгоритм помечает все вершины, связанные с исходной вершиной s (но не с остальными). Каждая помеченная вершина соединена с s, т.к. алгоритм попадает в вершины, только проходя по ребрам. Теперь предположим, что какая-то непомеченная вершина w соединена с s. Поскольку сама s помечена, любой путь из s в w должен содержать хотя бы одно ребро из множества помеченных вершин во множество непомеченных вершин — скажем, v-x. Но алгоритм должен был пройти в x после пометки v, поэтому такое ребро не существует, что противоречит исходному предположению (рис. 4.1.16). Временная граница следует из этого рассуждения, т.к. пометки гарантируют, что каждая вершина посещается только один раз (что требует времени на проверку пометок, пропорционального степени этой вершины).

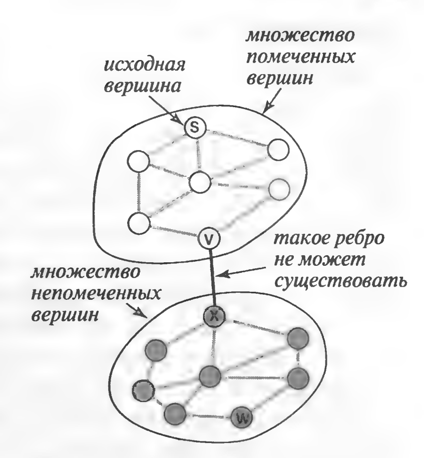


Рис. 4.1.16. К доказательству утверждения А

Односторонние проходы

Метод вызовов и возвратов в программах соответствует нити в лабиринте: после обследования всех ребер, инцидентных вершине (просмотрели все проходы, ведущие из пересечения), мы "выполняем возврат" — в обоих смыслах этого слова. Чтобы получить более четкое соответствие с правилом Тремо, необходимо представить, что лабиринт состоит исключительно из односторонних проходов (по одному в каждом направлении). Так же, как мы проходим по каждому проходу в лабиринте два раза (по одному в каждом направлении), мы проходим *дважды* и по каждому ребру в графе (из каждой его вершины и назад). При обходе по правилу Тремо мы либо движемся по проходу в первый раз, либо возвращаемся по нему от помеченной вершины; при ПвГ на неориентированном графе мы либо выполняем рекурсивный вызов при встрече с ребром v-w (если вершина w не помечена), либо пропускаем это ребро (если w помечена). Когда это ребро встретится еще раз, в противоположном направлении w-v, мы всегда игнорируем его, т.к. дальняя вершина v уже точно посещена (при первой встрече с ребром).

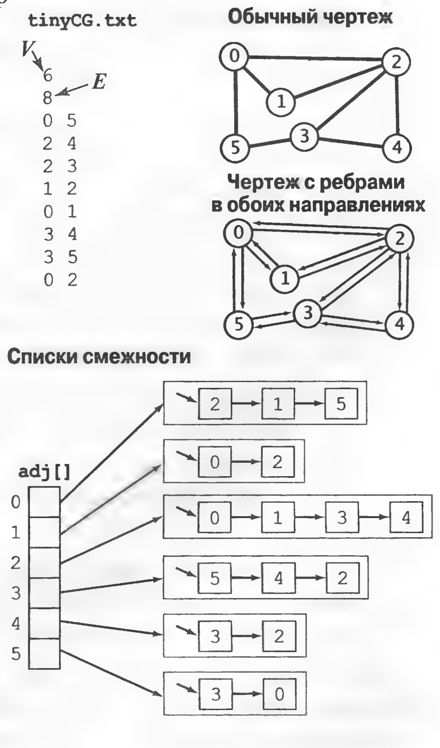


Рис. 4.1.17. Связный неориентированный граф

Трассировка ПвГ

Как обычно, хорошо помогает понять работу алгоритма знакомство с трассировкой его действий на небольшом примере. Это особенно верно в случае поиска в глубину. Первое, о чем нужно помнить при выполнении трассировки — что порядок посещения ребер и вершин зависит от *представления*, а не от графа или алгоритма. Поскольку ПвГ просматривает только вершины, связанные с исходной вершиной, мы воспользуемся в качестве примера для трассировки небольшим связным графом, приведенным на рис. 4.1.17. В этом примере после вершины 0 первой посещается вершина 2, т.к. она первая в списке смежности вершины 0. Второе, о чем нужно помнить при выполнении трассировки — что, как уже было сказано, ПвГ проходит по каждому ребру графа два раза, и во второй раз всегда обнаруживает помеченную вершину. То есть выполнение ПвГ занимает в два раза больше времени, чем вы могли подумать! В нашем демонстрационном примере всего восемь ребер, а нам приходится прослеживать действие алгоритма на 16 элементах списков смежности.

Подробная трассировка поиска в глубину

На рис. 4.1.18 показано содержимое структур данных сразу после пометки каждой вершины в нашем небольшом примере, начиная с вершины 0. Поиск начинается тогда, когда конструктор выполняет рекурсивный вызов `dfs()`, который помечает и посещает вершину 0 и далее делает следующее.

- Поскольку первой в списке связности вершины 0 находится вершина 2, метод `dfs()` рекурсивно вызывает себя для пометки и посещения 2 (при этом система помещает в стек вершину 0 и текущую позицию в списке связности вершины 0).
- Теперь первой в списке связности 2 находится вершина 0, но она уже помечена, поэтому `dfs()` пропускает ее. Следующей идет вершина 1, и она не помечена, поэтому `dfs()` рекурсивно вызывает себя, чтобы пометить и посетить 1.
- Посещение вершины 1 происходит по-другому: обе вершины в ее списке (0 и 2) уже помечены, поэтому рекурсивные вызовы не нужны, и `dfs()` выполняет возврат из вызова `dfs(1)`. Следующим просматривается ребро 2-3 (т.к. вершина 3 следует за 1 в списке смежности вершины 2), поэтому `dfs()` рекурсивно вызывает себя, чтобы пометить и посетить вершину 3.
- В списке смежности 3 первой находится вершина 5, и она не помечена, поэтому `dfs()` рекурсивно вызывает себя, чтобы пометить и посетить вершину 5.
- Обе вершины в списке 5 (3 и 0) уже помечены, поэтому `dfs()` рекурсивно вызывает себя, чтобы пометить и посетить вершину 4 — последнюю непомеченную вершину.
- После пометки 4 метод `dfs()` должен проверить вершины в ее списке, потом оставшиеся вершины в списке 3, потом в списке 2, потом в списке 0, но рекурсивные вызовы не выполняются, т.к. все вершины уже помечены.

Эта базовая рекурсивная схема — лишь начало, т.к. поиск в глубину позволяет эффективно решать многие задачи обработки графов. Например, в данном разделе мы рассмотрим применение поиска в глубину для решения задачи, сформулированной еще в главе 1.

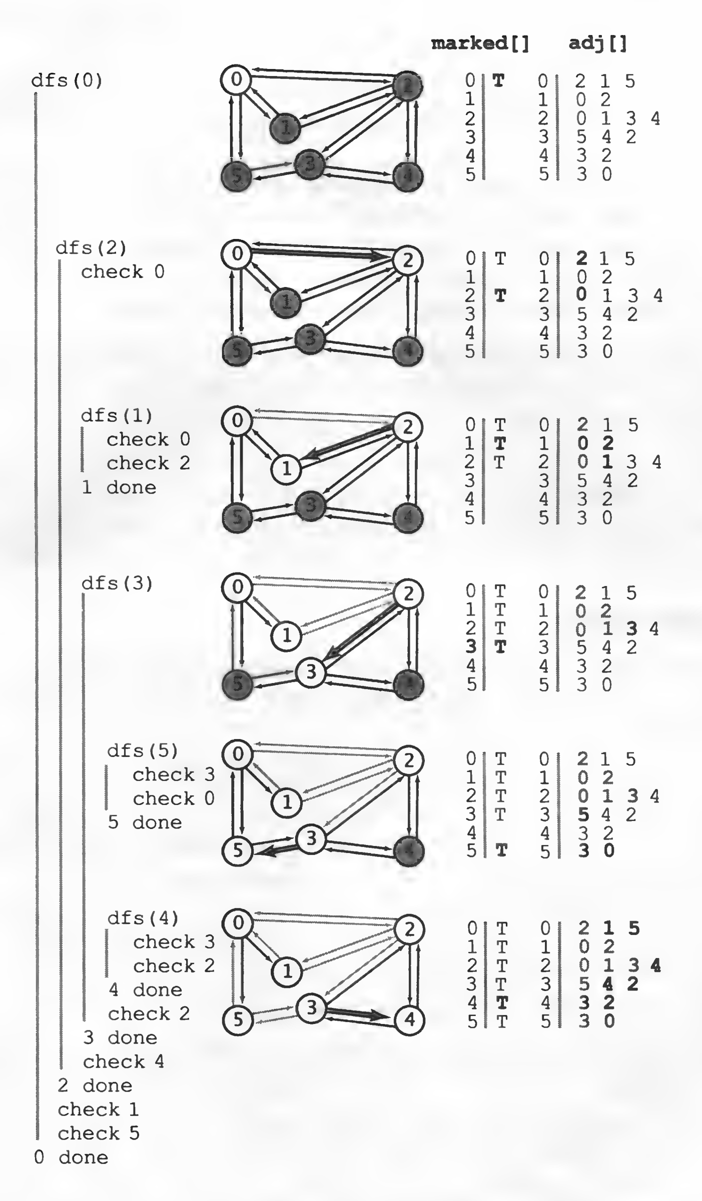


Рис. 4.1.18. Трассировка поиска в глубину для нахождения вершин, связанных с **0**

Связность. Для заданного графа нужны ответы на запросы вида *Связаны ли две указанные вершины?* и *Сколько связных компонентов содержит граф?*

Эта задача легко решается с помощью нашего шаблона обработки графов, и мы сравним полученное решение с алгоритмами объединения-поиска, которые были рассмотрены в разделе 1.5.

Вопрос "Связаны ли две указанные вершины" эквивалентен вопросу "Существует ли путь, соединяющий две указанные вершины?", и его можно назвать задачей *нахождения пути*. Однако структуры данных объединения-поиска из раздела 1.5 не могут решать задачи *нахождения* такого пути. Поиск в глубину — первый способ, который мы рассмотрим в связи с решением данной задачи, а также следующей задачи.

Пути из одного источника. Для заданного графа и исходной вершины s нужны ответы на запросы вида *Существует ли путь из* s *до указанной вершины* v*? Если да, то нужно найти такой путь.*

Алгоритм ПвГ с виду прост, т.к. он основан на понятной концепции и легко реализуется. Однако это тонкий и мощный алгоритм, который позволяет решать многочисленные сложные задачи. Сформулированные выше задачи — первые из тех, которые мы рассмотрим.

Нахождение путей

Задача путей из одного источника является фундаментальной задачей в обработке графов. В соответствии с нашей стандартной схемой построения мы воспользуемся API, приведенным на рис. 4.1.19.

public class **Paths**		
	Paths(Graph G, int s)	*поиск путей в G из источника s*
boolean	hasPathTo(int v)	*существует ли путь из s в v?*
Iterable<Integer>	pathTo(int v)	*путь из s в v; нулевой объект, если путь не существует*

Рис. 4.1.19. *API для реализаций путей*

Конструктор принимает в качестве аргумента исходную вершину s и вычисляет пути из s в каждую вершину, связанную с s. После создания объекта Paths для источника s клиент может использовать метод экземпляров pathTo() для перебора вершин пути от s до любой вершины, связанной с s. Пока нам годятся любые пути, а позже мы будем разрабатывать реализации, которые находят пути с определенными свойствами. Клиент тестирования, приведенный в листинге 4.1.4, вводит граф из входного потока и источник из командной строки, а затем выводит пути из источника до каждой связанной с ним вершины.

Листинг 4.1.4. Клиент тестирования для реализаций путей

```
public static void main(String[] args)
{
   Graph G = new Graph(new In(args[0]));
   int s = Integer.parseInt(args[1]);
```

```
    Paths search = new Paths(G, s);
    for (int v = 0; v < G.V(); v++)
    {
      StdOut.print(s + " to " + v + ": ");
      if (search.hasPathTo(v))
        for (int x : search.pathTo(v))
          if (x == s) StdOut.print(x);
          else StdOut.print("-" + x);
      StdOut.println();
    }
}
```

```
% java Paths tinyCG.txt 0
0 to 0: 0
0 to 1: 0-2-1
0 to 2: 0-2
0 to 3: 0-2-3
0 to 4: 0-2-3-4
0 to 5: 0-2-3-5
```

Реализация

Алгоритм 4.1, приведенный в листинге 4.1.5, представляет собой реализацию API Paths на основе ПвГ и расширяет наш разминочный класс DepthFirstSearch из листинга 4.1.3. В нем добавлен массив edgeTo[] значений int, который играет роль нити в правиле обхода Тремо и позволяет вернуться в s из каждой связанной с ней вершины. Здесь запоминается не просто путь из текущей вершины до исходной, а путь из *каждой* вершины до исходной. Для этого мы запоминаем ребро v-w, которое *впервые* привело нас в каждую вершину w, занося в edgeTo[w] значение v. То есть v-w — последнее ребро на известном пути из s в w. В результате поиска образуется дерево с корнем в исходной вершине, а edgeTo[] содержит представление этого дерева родительскими ссылками. Небольшой пример см. на рис. 4.1.20. Чтобы восстановить путь из s до любой вершины v, метод pathTo() из алгоритма 4.1 возвращается вверх по дереву с помощью переменной x, присваивая ей значение edgeTo[x]. Аналогично вел себя алгоритм объединения-поиска из раздела 1.5, где каждая вершина заносилась в стек до попадания в s. Возврат этого стека клиенту в виде Iterable позволяет клиенту пройти путь от s до v.

Листинг 4.1.5. Алгоритм 4.1. Поиск в глубину для нахождения путей в графе

```
public class DepthFirstPaths
{
  private boolean[] marked;   // Вызывался ли dfs() для этой вершины?
  private int[] edgeTo; // Последняя вершина на известном пути до данной вершины
  private final int s;        // Исходная вершина
  public DepthFirstPaths(Graph G, int s)
  {
    marked = new boolean[G.V()];
    edgeTo = new int[G.V()];
    this.s = s;
    dfs(G, s);
  }
```

```
    private void dfs(Graph G, int v)
    {
       marked[v] = true;
       for (int w : G.adj(v))
          if (!marked[w])
          {
             edgeTo[w] = v;
             dfs(G, w);
          }
    }
    public boolean hasPathTo(int v)
    {  return marked[v];  }
    public Iterable<Integer> pathTo(int v)
    {
       if (!hasPathTo(v)) return null;
       Stack<Integer> path = new Stack<Integer>();
       for (int x = v; x != s; x = edgeTo[x])
          path.push(x);
       path.push(s);
       return path;
    }
}
```

edgeTo[]

0	
1	2
2	0
3	2
4	3
5	3

x	path
5	5
3	3 5
2	2 3 5
0	0 2 3 5

Рис. 4.1.20. Трассировка вычисления `pathTo(5)`

Эта реализация Graph использует поиск в глубину, чтобы находить пути в связном графе до всех вершин от заданной вершины s. Код из класса DepthFirstSearch (листинг 4.1.3) выделен серым шрифтом. Для сохранения известных путей к каждой вершине используется индексированный вершинами массив edgeTo[], такой, что edgeTo[w] = v означает, что впервые алгоритм добрался до вершины w по ребру v-w. Массив edgeTo[] — представление родительскими ссылками для дерева с корнем в вершине s, которое содержит все вершины, связанные с s.

Подробная трассировка

На рис. 4.1.21 показано содержимое массива edgeTo[] сразу после пометки каждой вершины в нашем примере с исходной вершиной 0. Содержимое массивов marked[] и adj[] совпадает с трассировкой класса DepthFirstSearch (рис. 4.1.18), совпадает и подробное описание рекурсивных вызовов и пометок ребер — поэтому эти аспекты трассировки опущены. Поиск в глубину добавляет в массив edgeTo[] ребра 0-2, 2-1, 2-3, 3-5 и 3-4 в указанном порядке. Эти ребра формируют дерево с корнем в исходной вершине и содержат информацию, необходимую методу pathTo(), когда ему нужно передать клиенту путь от вершины 0 к вершине 1, 2, 3, 4 или 5.

Утверждение А (продолжение). Поиск в глубину позволяет выдавать клиентам путь из указанного источника к любой помеченной вершине за время, пропорциональное его длине.

Доказательство. По индукции по количеству посещенных вершин следует, что массив edgeTo[] в классе DepthFirstPaths представляет дерево с корнем в исходной вершине. Метод pathTo() строит путь за время, пропорциональное его длине.

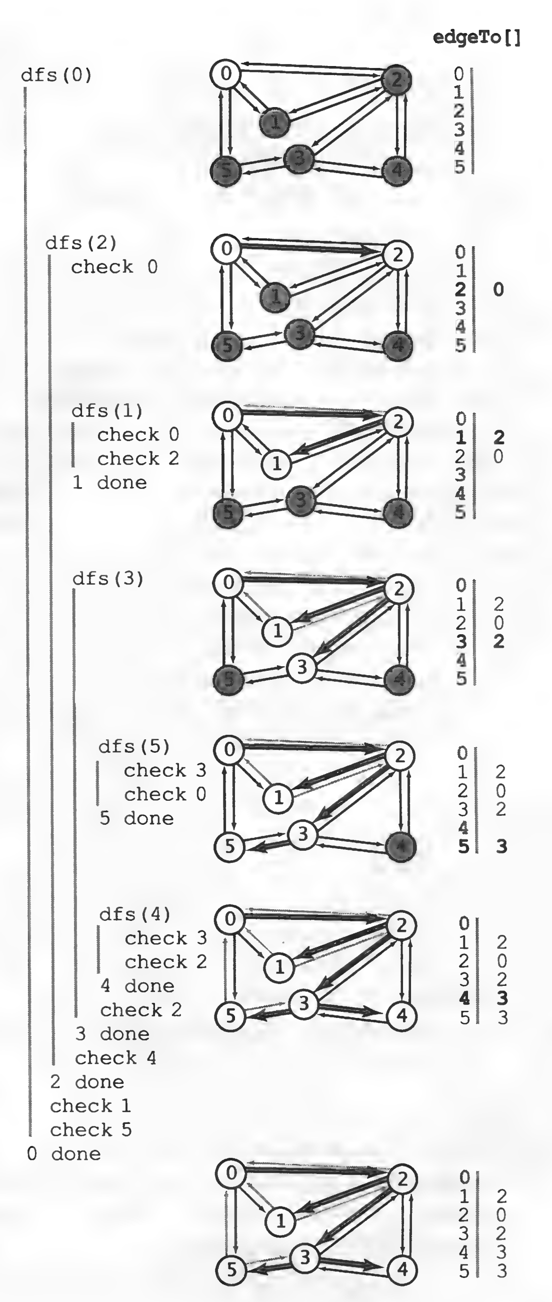
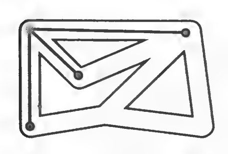


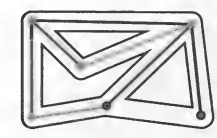
*Рис. 4.1.21. Трассировка поиска в глубину для нахождения всех путей из **0***

Поиск в ширину

Пути, найденные поиском в глубину, зависят не только от графа, но и от его представления, и от природы рекурсии. Зачастую естественно возникает следующая задача.

Кратчайшие пути из одного источника. Для заданного графа и исходной вершины s нужны ответы на запросы вида *Существует ли путь из* s *к указанной вершине* v*?* Если да, то нужно найти *кратчайший* такой путь (с минимальным количеством ребер).





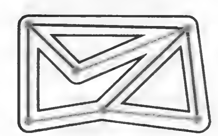


Рис. 4.1.22. Проход по лабиринту по правилу "в ширину"

Классический способ выполнения таких задач называется *поиском в ширину* (ПвШ). На нем основаны многочисленные алгоритмы обработки графов, и поэтому мы подробно изучим его в данном разделе. ПвГ мало чем может помочь нам в этом вопросе, поскольку порядок, в котором он проходит по графу, никак не соотносится с целью нахождения кратчайших путей. А вот ПвШ как раз предназначен для этого. Чтобы найти кратчайший путь из s в v, мы начинаем с вершины s и проверяем, есть ли вершина v среди всех вершин, до которых можно добраться, пройдя по одному ребру, потом проверяем, есть ли вершина v среди всех вершин, до которых можно добраться, пройдя по двум ребрам, и т.д. (рис. 4.1.22). ПвГ похож на исследование лабиринта одним человеком, а ПвШ аналогичен группе исследователей, которые разбредаются по всем направлениям, и каждый разматывает собственный клубок. Если нужно исследовать более одного прохода, исследователи как бы разделяются, чтобы их хватило на все проходы; а когда две группы исследователей встречаются, они объединяются (используя клубок того, кто первый попал в место встречи).

Когда в программе мы попадаем в точку, из которой нужно пройти более чем по одному ребру, мы выбираем одно ребро, а остальные запоминаем. В ПвГ для этой цели используется стек (управляемый системой для поддержки метода рекурсивного поиска). Использование правила LIFO, характерного для стека, соответствует исследованию проходов, которые находятся ближе всего в лабиринте. Мы выбираем из всех еще не исследованных проходов тот, который обнаружили последним. В ПвШ вершины нужно исследовать в порядке их удаления от источника: вместо стека (LIFO) используется очередь (FIFO). Из всех еще не исследованных проходов выбирается обнаруженный раньше всего.

Реализация

Алгоритм 4.2 из листинга 4.1.6 представляет собой реализацию ПвШ. В нем используется очередь всех вершин, которые уже помечены, но списки смежности которых еще не просмотрены. Мы помещаем в очередь исходную вершину, а затем выполняем следующие шаги, пока очередь не станет пустой:

- выбираем из очереди следующую вершину v и помечаем ее;
- помещаем в очередь все непомеченные вершины, смежные с v.

Листинг 4.1.6. Алгоритм 4.2. Поиск в ширину для нахождения путей в графе

```
public class BreadthFirstPaths
{
  private boolean[] marked;    // Известен ли кратчайший путь к этой вершине?
  private int[] edgeTo;  // Последняя вершина на известном пути к данной вершине
  private final int s;         // Исходная вершина

  public BreadthFirstPaths(Graph G, int s)
  {
    marked = new boolean[G.V()];
    edgeTo = new int[G.V()];
    this.s = s;
    bfs(G, s);
  }

  private void bfs(Graph G, int s)
  {
    Queue<Integer> queue = new Queue<Integer>();
    marked[s] = true;          // Пометка исходной вершины
    queue.enqueue(s);          //   и помещение ее в очередь.
    while (!q.isEmpty())
    {
      int v = queue.dequeue();// Извлечение из очереди следующей вершины.
      for (int w : G.adj(v))
        if (!marked[w])        // Для каждой непомеченной смежной вершины:
        {
          edgeTo[w] = v;       //   сохраняем последнее ребро в кратчайшем пути,
          marked[w] = true;    //   помечаем ее, т.к. путь известен,
          queue.enqueue(w);    //   и заносим ее в очередь.
        }
    }
  }

  public boolean hasPathTo(int v)
  {  return marked[v];  }

  public Iterable<Integer> pathTo(int v)
  // Совпадает с поиском в глубину (см. алгоритм 4.1).
}
```

Этот клиент Graph использует поиск в ширину, чтобы находить пути в графе с наименьшим количеством ребер из источника s, указанного в конструкторе. Метод bfs() помечает все вершины, связанные с s, и клиент может использовать метод hasPathTo() для определения, связана ли указанная вершина v с вершиной s, и метод pathTo(), чтобы получить путь из s в v, такой, что никакой другой путь из s в v не содержит меньшее количество ребер.

```
% java BreadthFirstPaths tinyCG.txt 0
0 to 0: 0
0 to 1: 0-1
0 to 2: 0-2
0 to 3: 0-2-3
0 to 4: 0-2-4
0 to 5: 0-5
```

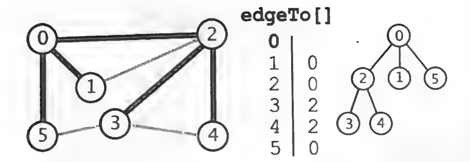


Рис. 4.1.23. Результат поиска в ширину для нахождения всех путей из 0

Метод `bfs()` в алгоритме 4.2 *не* рекурсивен. Вместо неявного стека, предоставляемого рекурсией, в нем применяется явная очередь. Результатом поиска, как и в ПвГ, является массив `edgeTo[]` — представление родительскими ссылками дерева с корнем в `s`, которое определяет все кратчайшие пути из `s` в каждую вершину, связанную с `s` (рис. 4.1.23). Клиент может построить нужный путь с помощью такой же реализации `pathTo()`, что и для ПвГ из алгоритма 4.1.

На рис. 4.1.24 показан пошаговый процесс работы ПвШ на нашем демонстрационном графе, с содержимым структур данных в начале каждой итерации цикла. Сначала в очередь помещается вершина 0, а затем цикл завершает поиск, выполняя следующие действия.

queue: 0

	marked[]	edgeTo[]	adj[]
0	T		2 1 5
1			0 2
2			0 1 3 4
3			5 4 2
4			3 2
5			3 0

queue: 2 1 5

	marked[]	edgeTo[]	adj[]
0	T		2 1 5
1	T	0	0 2
2	T	0	0 1 3 4
3			5 4 2
4			3 2
5	T	0	3 0

queue: 1 5 3 4

	marked[]	edgeTo[]	adj[]
0	T		2 1 5
1	T	0	0 2
2	T	0	0 1 3 4
3	T	2	5 4 2
4	T	2	3 2
5	T	0	3 0

queue: 5 3 4

	marked[]	edgeTo[]	adj[]
0	T		2 1 5
1	T	0	0 2
2	T	0	0 1 3 4
3	T	2	5 4 2
4	T	2	3 2
5	T	0	3 0

queue: 3 4

	marked[]	edgeTo[]	adj[]
0	T		2 1 5
1	T	0	0 2
2	T	0	0 1 3 4
3	T	2	5 4 2
4	T	2	3 2
5	T	0	3 0

queue: 4

	marked[]	edgeTo[]	adj[]
0	T		2 1 5
1	T	0	0 2
2	T	0	0 1 3 4
3	T	2	5 4 2
4	T	2	3 2
5	T	0	3 0

Рис. 4.1.24. Трассировка поиска в ширину для нахождения всех путей из 0

- Из очереди удаляется вершина 0, и в очередь заносятся смежные с 0 вершины 2, 1 и 5. Каждая из них помечается, а в соответствующие элементы edgeTo[] заносится 0.
- Из очереди удаляется вершина 2, проверяются смежные с ней вершины 0 и 1 (они уже помечены), и в очередь заносятся смежные вершины 3 и 4. Каждая из них помечается, а в соответствующие элементы edgeTo[] заносится 2.
- Из очереди удаляется вершина 1, и проверяются смежные с ней вершины 0 и 2 — они уже помечены.
- Из очереди удаляется вершина 5, и проверяются смежные с ней вершины 3 и 0 — они уже помечены.
- Из очереди удаляется вершина 3, и проверяются смежные с ней вершины 5, 4 и 2 — они уже помечены.
- Из очереди удаляется вершина 4, и проверяются смежные с ней вершины 3 и 2 — они уже помечены.

В этом примере массив edgeTo[] оказывается полностью оформленным после второго шага. Как и в ПвГ, после пометки всех вершин вычисление только проверяет, что ребра ведут в уже помеченные вершины.

Утверждение Б. Для каждой вершины v, достижимой из s, ПвШ вычисляет кратчайший путь из s в v (ни один путь из s в v не содержит меньше ребер).

Доказательство. Нетрудно доказать по индукции, что очередь всегда состоит из ноля или более вершин, которые находятся на расстоянии k от источника, за которыми следуют ноль или более вершин на расстоянии $k+1$ от источника — для некоторого k, начиная с $k = 0$. Из этого, в частности, следует, что вершины заносятся в очередь и удаляются из нее в порядке их расстояния от s. Когда вершина v заносится в очередь, уже не будет найден более короткий путь до v, прежде чем она будет удалена из очереди, и ни один путь до v, обнаруженный после удаления ее из очереди не будет короче, чем путь по уже сформированному дереву.

Утверждение Б (продолжение). Поиск в ширину в худшем случае требует времени, пропорционального $V+E$.

Доказательство. Как и в утверждении А, поиск в ширину помечает все вершины, связанные с s, за время, пропорциональное сумме их степеней. Если граф связен, эта сумма равна сумме степеней всех его вершин, т.е. $2E$.

ПвШ также можно использовать для реализации API Search, для которого мы использовали ПвГ, потому что решение зависит только от возможности поиска просмотреть каждую вершину и ребро, связанное с вершиной.

Как уже было сказано, ПвГ и ПвШ — первые из нескольких методов, которые мы еще будем рассматривать для поиска на графах. Мы помещаем исходную вершину в структуру данных, а затем выполняем следующие шаги, пока структура данных не станет пустой:

- выбираем из структуры данных следующую вершину v и помечаем ее;
- помещаем в структуру данных все непомеченные вершины, смежные с v.

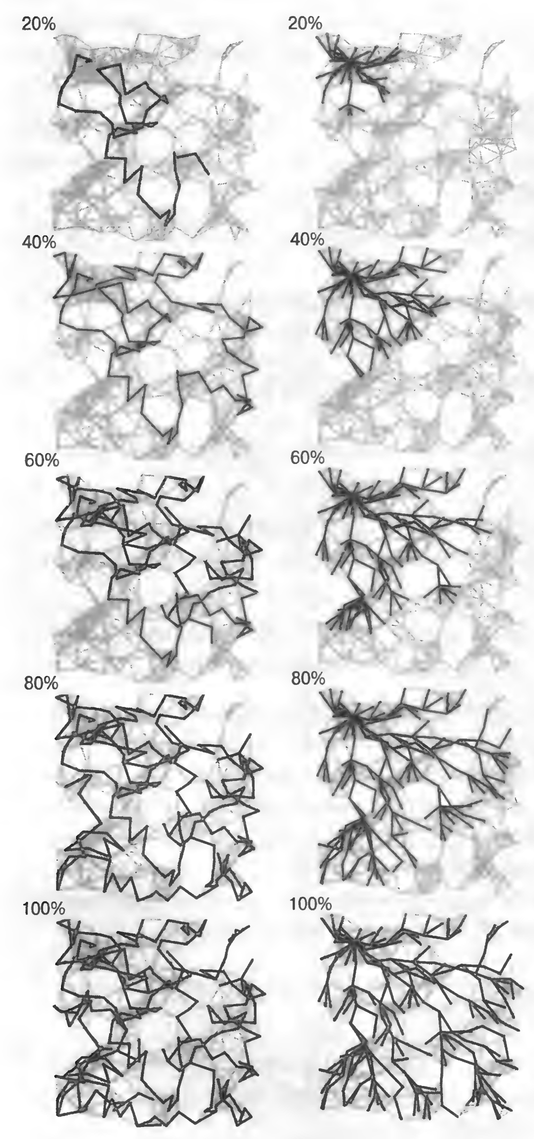


Рис. 4.1.25. ПвГ для путей (250 вершин)

Рис. 4.1.26. ПвШ для кратчайших путей (250 вершин)

Алгоритмы отличаются только правилом выборки следующей вершины из структуры данных (раньше всех добавленную в ПвШ и позже всех добавленную в ПвГ). Эта разница приводит к совершенно разным представлениям графа, хотя все вершины и ребра, связанные с источником, просматривается независимо от правила.

На рис. 4.1.25 и 4.1.26 показано выполнение ПвГ и ПвШ для нашего демонстрационного графа `mediumG.txt` — хорошо видно различие между путями, обнаруживаемыми этими двумя способами. ПвГ прокладывает путь вглубь графа, сохраняя в стеке точки, где замечены ответвления; а ПвШ расползается по всему графу, используя очередь для запоминания границы посещенных мест. ПвГ исследует граф, находя новые вершины все дальше от исходной точки и посещая более близкие только после обнаружения тупиков; а ПвШ полностью охватывает область вблизи начальной точки, уходя дальше только после обследования ближайшей окрестности. Пути ПвГ обычно длинные и изогнутые, а пути ПвШ короткие и прямые. В зависимости от приложения, может понадобиться как то, так и другое поведение (хотя может быть и так, что свойства путей окажутся не важными). В разделе 4.4 мы познакомимся и с другими реализациями API `Paths`, которые находят пути с другими свойствами.

Связные компоненты

Теперь мы рассмотрим другое применение поиска в глубину — поиск связных компонентов в графе. В разделе 1.5 было сказано, что свойство "связан" является *отношением эквивалентности*, которое разбивает вершины на *классы эквивалентности* (связные компоненты). Для этой часто встречающейся задачи обработки графов мы определим специальный API (рис. 4.1.27).

Метод `id()` позволяет проиндексировать массив номерами компонентов — как в клиенте тестирования из листинга 4.1.7, который читает граф, потом выводит номера его связных компонентов, а затем выводит вершины из каждого компонента, по одному компоненту в строке. Для этого он создает массив объектов `Bag` и использует идентификаторы компонентов каждой вершины в качестве индексов в этом массиве, чтобы зано-

сить вершины в соответствующие объекты Bag. Этот клиент представляет собой модель для типичных ситуаций, где нужна независимая обработка связных компонентов.

public class **CC**	
CC(Graph G)	*конструктор с предобработкой*
boolean connected(int v, int w)	*связаны ли v и w?*
int count()	*количество связных компонентов*
int id(int v)	*идентификатор компонента для v (от 0 до count()-1)*

Рис. 4.1.27. *API для связных компонентов*

Листинг 4.1.7. Клиент тестирования для API связных компонентов

```
public static void main(String[] args)
{
   Graph G = new Graph(new In(args[0]));
   CC cc = new CC(G);
   int M = cc.count();
   StdOut.println(M + " компонентов");
   Bag<Integer>[] components;
   components = (Bag<Integer>[]) new Bag[M];
   for (int i = 0; i < M; i++)
      components[i] = new Bag<Integer>();
   for (int v = 0; v < G.V(); v++)
      components[cc.id(v)].add(v);
   for (int i = 0; i < M; i++)
   {
      for (int v: components[i])
         StdOut.print(v + " ");
      StdOut.println();
   }
}
```

Реализация

Реализация API CC (алгоритм 4.3 в листинге 4.1.8, см. также рис. 4.1.28) использует массив marked[], чтобы найти вершину, которая может служить начальной точкой для поиска в глубину в каждом компоненте. Первый вызов рекурсивного ПвГ выполняется для вершины 0, и он помечает значением 0 все связанные с ней вершины. Затем цикл for в конструкторе ищет еще не помеченную вершину и вызывает рекурсивный метод dfs(), чтобы пометить все вершины, связанные с этой вершиной. Кроме того, в нем задействован индексированный вершинами массив id[], который связывает с каждой вершиной в каждом отдельном компоненте одно и тоже целочисленное значение. Этот массив делает реализацию метода connected() тривиальной (точно так же, как и метод connected() из раздела 1.5) — достаточно сравнить идентификаторы компонентов для вершин. В данном случае значение 0 присваивается всем вершинам в первом обработанном компоненте, значение 1 — во втором обработанном компоненте и т.д., и эти идентификаторы находятся в диапазоне от 0 до count()-1, как и указано в API. Это соглашение позволяет использовать индексированные компонентами массивы, как в клиенте тестирования из листинга 4.1.7.

Листинг 4.1.8. Алгоритм 4.3. Поиск в глубину для нахождения связных компонентов в графе

```
public class CC
{
  private boolean[] marked;
  private int[] id;
  private int count;

  public CC(Graph G)
  {
    marked = new boolean[G.V()];
    id = new int[G.V()];
    for (int s = 0; s < G.V(); s++)
      if (!marked[s])
      {
         dfs(G, s);
         count++;
      }
  }

  private void dfs(Graph G, int v)
  {
    marked[v] = true;
    id[v] = count;
    for (int w : G.adj(v))
      if (!marked[w])
         dfs(G, w);
  }

  public boolean connected(int v, int w)
  {  return id[v] == id[w];  }

  public int id(int v)
  {  return id[v];  }

  public int count()
  {  return count;  }
}
```

Этот клиент Graph позволяет клиентам независимо обрабатывать связные компоненты графа. Код, перенесенный из класса DepthFirstSearch (листинг 4.1.3), выделен серым шрифтом. Вычисление использует индексированный вершинами массив id[], такой, что значение id[v] равно i, если v принадлежит i-тому связному компоненту из числа обработанных. Конструктор находит непомеченную вершину и вызывает рекурсивный метод dfs(), который помечает все связанные с ней вершины, и продолжает так до тех пор, пока не будут помечены все вершины. Реализации методов экземпляров connected(), id() и count() очевидны.

```
% more tinyG.txt
13 вершин, 13 ребер
0: 6 2 1 5
1: 0
2: 0
3: 5 4
4: 5 6 3
```

```
5: 3 4 0
6: 0 4
7: 8
8: 7
9: 11 10 12
10: 9
11: 9 12
12: 11 9
% java CC tinyG.txt
3 компонентов
6 5 4 3 2 1 0
8 7
12 11 10 9
```

Утверждение В. Поиск в глубину использует время на предобработку и объем памяти, пропорциональные $V+E$, а затем может отвечать на запросы связности в графе за константное время.

Доказательство. Следует непосредственно из кода. Каждый элемент списка смежных вершин проверяется в точности один раз, и существует $2E$ таких элементов (по два для каждого ребра). Методы экземпляров проверяют или возвращают одну или две переменные экземпляров.

Объединение-поиск

Как решение задачи связности для графа в классе CC на основе ПвГ выглядит в сравнении с методом объединения-поиска из главы 1? Теоретически ПвГ быстрее, чем объединение-поиск, поскольку он, в отличие от объединения-поиска, гарантирует ответы на запросы за константное время. Но на практике это отличие незаметно, а объединение-поиск работает быстрее, т.к. ему не нужно строить полное представление графа. Что более важно, объединение-поиск является оперативным алгоритмом: он позволяет проверить, связаны ли две вершины, за почти константное время в любой момент, даже в процессе добавления ребер, а решение на базе ПвГ должно сначала выполнить предобработку графа. Поэтому мы считаем, что объединение-поиск удобнее в тех случаях, когда нужно только определить связность или когда нужно отвечать на большое количество запросов вперемешку со вставками ребер, а решение на основе ПвГ удобнее в АТД графа, потому что оно эффективно использует существующую инфраструктуру.

Задачи, которые можно решить с помощью ПвГ — фундаментальные задачи. Это простой подход, а рекурсия позволяет анализировать вычисления и разрабатывать компактные решения для задач обработки графов. В табл. 4.1.3 приведены еще два примера для решения следующих задач.

Обнаружение циклов. Ответы на запрос: *Содержит ли указанный граф циклы?*

Двухцветность. Ответы на запрос: *Можно ли каждой вершине графа присвоить один из двух цветов, чтобы ни одно ребро не соединяло две вершины одного цвета?* Это эквивалентно вопросу *Является ли граф двудольным?*

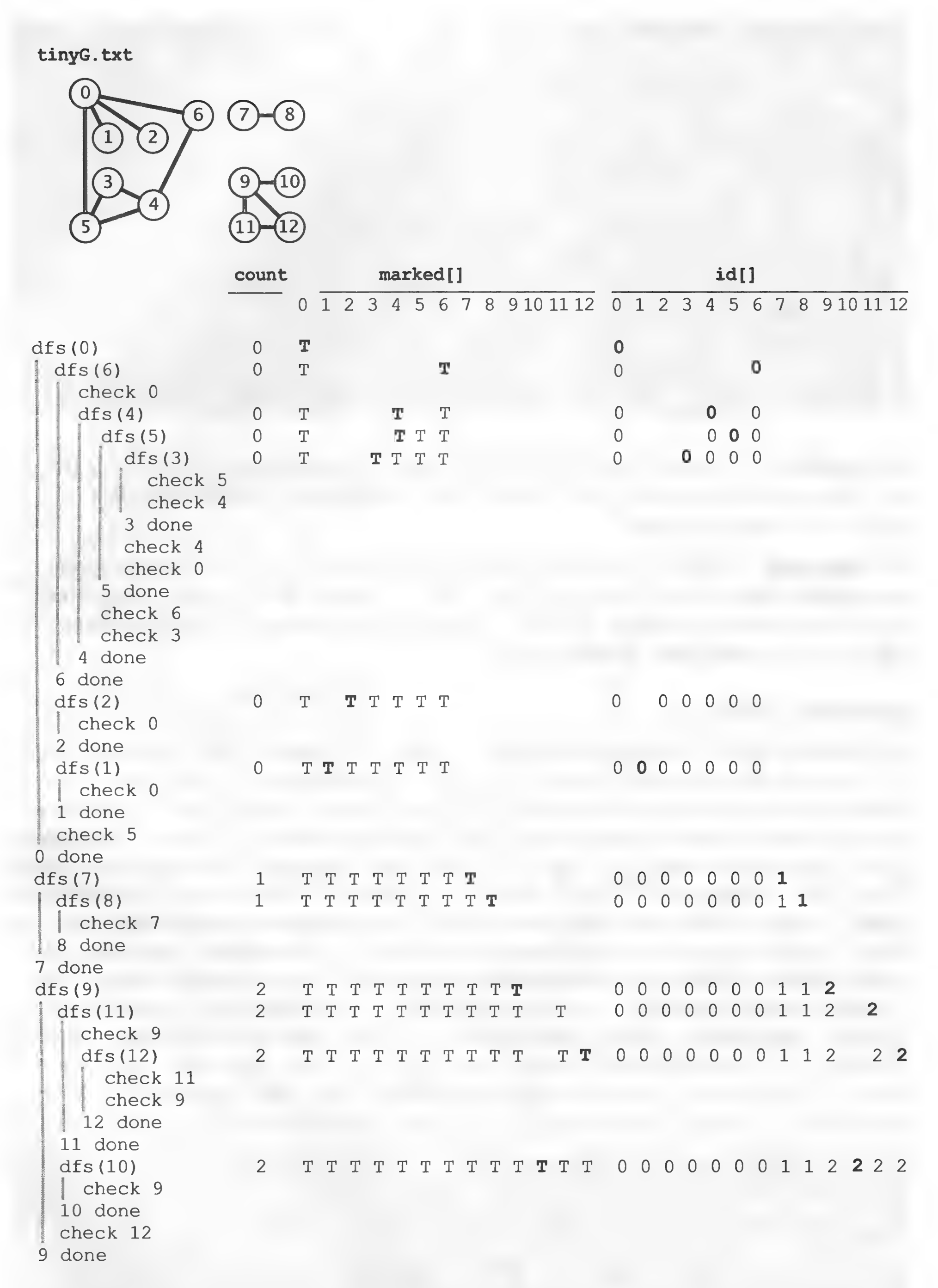


Рис. 4.1.28. Трассировка поиска в глубину для нахождения связных компонентов

Таблица 4.1.3. Дополнительные примеры обработки графов с помощью ПвГ

Задача	Реализация
Содержит ли G циклы? (Предполагается отсутствие петель и параллельных ребер.)	`public class Cycle { private boolean[] marked; private boolean hasCycle; public Cycle(Graph G) { marked = new boolean[G.V()]; for (int s = 0; s < G.V(); s++) if (!marked[s]) dfs(G, s, s); } private void dfs(Graph G, int v, int u) { marked[v] = true; for (int w : G.adj(v)) if (!marked[w]) dfs(G, w, v); else if (w != u) hasCycle = true; } public boolean hasCycle() { return hasCycle; } }`
Является ли G двудольным? (Допускает ли раскраску двумя цветами?	`public class TwoColor { private boolean[] marked; private boolean[] color; private boolean isTwoColorable = true; public TwoColor(Graph G) { marked = new boolean[G.V()]; color = new boolean[G.V()]; for (int s = 0; s < G.V(); s++) if (!marked[s]) dfs(G, s); } private void dfs(Graph G, int v) { marked[v] = true; for (int w : G.adj(v)) if (!marked[w]) { color[w] = !color[v]; dfs(G, w); } else if (color[w] == color[v]) isTwoColorable = false; } public boolean isBipartite() { return isTwoColorable; }`

Как обычно в случае ПвГ, простота кода маскирует более сложные вычисления, поэтому рекомендуется изучить (в упражнениях) данные примеры, проследить их поведение на простых примерах графов и на расширениях графов с циклами и несколькими цветами соответственно.

Символьные графы

В типичных приложениях выполняется обработка графов, которые определены в файлах или веб-страницах, и вершины в них задаются строками, а не целочисленными индексами. Для работы с такими приложениями мы определим формат входных данных с перечисленными ниже свойствами.

- Имена вершин являются строками.
- Имена вершин разграничиваются указанным разделителем (чтобы имена могли содержать пробелы).
- Каждая строка представляет собой множество ребер, соединяющих первую вершину в этой строке с каждой из последующих вершин в этой же строке.
- Количество вершин V и количество ребер E определяются неявно.

На рис. 4.1.29 приведен пример — файл `routes.txt`, описывающий модель для небольшой транспортной системы, где вершины представляют собой коды аэропортов США, а соединяющие их ребра — авиарейсы, совершаемые между этими аэропортами. Этот файл содержит просто список ребер. На рис. 4.1.30 изображен пример побольше: он взят из файла `movies.txt` для Интернет-базы данных фильмов, с которой мы познакомились в разделе 3.5. Вспомните, что этот файл состоит из строк, и в каждой такой строке вначале идет название фильма, а за ним — список актеров, задействованных в этом фильме. В контексте обработки графов такой файл можно рассматривать как определение графа с фильмами и актерами в качестве вершин, где каждая строка определяет список смежности для ребер, соединяющих каждый фильм с его актерами. Обратите внимание, что этот граф является *двудольным*: в нем нет ребер, соединяющих актеров с актерами или фильмы с фильмами.

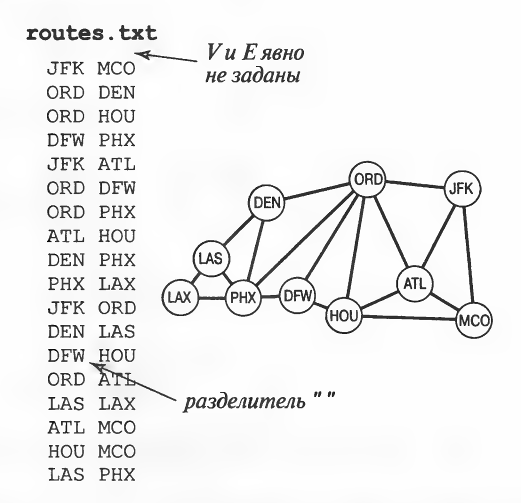
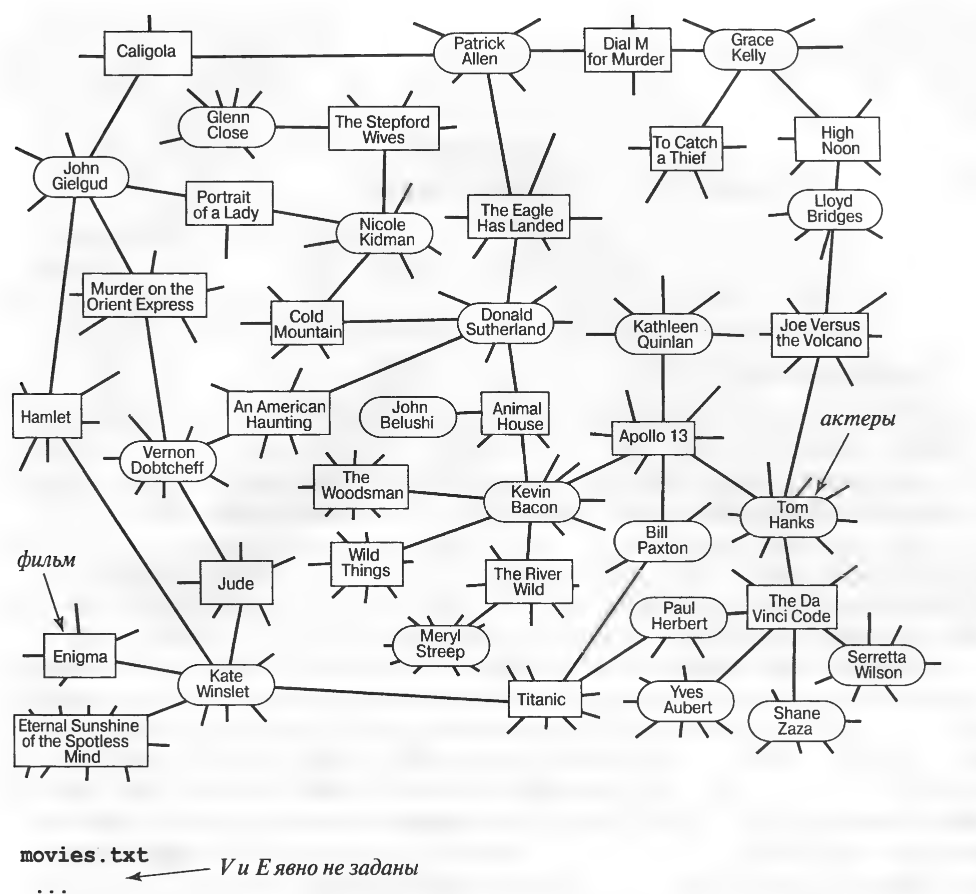


Рис. 4.1.29. Пример символьного графа (список ребер)




```
...
Tin Men (1987)/DeBoy, David/Blumenfeld, Alan/... /Geppi, Cindy/Hershey, Barbara...
Tirez sur le pianiste (1960)/Heymann, Claude/.../Berger, Nicole (I)...
Titanic (1997)/Mazin, Stan/...DiCaprio, Leonardo/.../Winslet, Kate/...
Titus (1999)/Weisskopf, Hermann/Rhys, Matthew/.../McEwan, Geraldine
To Be or Not to Be (1942)/Verebes, Ernö (I)/.../Lombard, Carole (I)...
To Be or Not to Be (1983)/.../Brooks, Mel (I)/.../Bancroft, Anne/...
To Catch a Thief (1955)/París, Manuel/.../Grant, Cary/.../Kelly, Grace/...
To Die For (1995)/Smith, Kurtwood/.../Kidman, Nicole/.../ Tucci, Maria...
...
```

Рис. 4.1.30. Пример символьного графа (списки смежности)

API

API, приведенный на рис. 4.1.31, определяет клиент класса `Graph`, который позволяет непосредственно использовать наши подпрограммы обработки графов для графов, определенных в таких файлах.

Этот API содержит конструктор, который читает и строит граф, и клиентские методы `name()` и `index()` для трансляции имен вершин из строк во входном потоке в целочисленные индексы, используемые методами обработки графов, и наоборот.

public class **SymbolGraph**	
SymbolGraph(String filename, String delim)	*создание графа, заданного в файле* filename, *где имена вершин разделяются символами* delim
boolean contains(String key)	*присутствует ли вершина* key?
int index(String key)	*индекс, связанный с* key
String name(int v)	*ключ, связанный с индексом* v
Graph G()	*базовая структура* Graph

***Рис. 4.1.31.** API для графов с символьными именами вершин*

Клиент тестирования

Клиент тестирования, приведенный в листинге 4.1.9, создает граф из файла, который указывается первым аргументом командной строки, с использованием разделителя, который указывается вторым аргументом, а затем принимает запросы из стандартного ввода. Пользователь задает имя вершины и получает список вершин, смежных с ней. Этот клиент непосредственно выполняет функции инвертированного индекса, который мы рассматривали в разделе 3.5. В случае файла routes.txt можно ввести код аэропорта и получить прямые рейсы из этого аэропорта — информацию, которую трудно получить непосредственно из исходных данных. В случае файла movies.txt можно ввести имя актера и получить список фильмов из базы данных, где засветился этот актер; или можно ввести название фильма и получить список актеров из этого фильма. Ввод названия фильма и получение его актерского состава сводится просто к возврату соответствующей строки из входного файла, но, чтобы ввести имя актера и получить список фильмов, где снимался этот актер, нужен инвертированный индекс. Хотя база данных построена на связи фильмов с актерами, модель двудольного графа связывает и актеров с фильмами. Модель двудольного графа автоматически служит в качестве инвертированного индекса, а также, как мы увидим, в качестве основы для более сложной обработки.

Листинг 4.1.9. Клиент тестирования для API символьного графа

```
public static void main(String[] args)
{
   String filename = args[0];
   String delim = args[1];
   SymbolGraph sg = new SymbolGraph(filename, delim);
   Graph G = sg.G();
   while (StdIn.hasNextLine())
   {
      String source = StdIn.readLine();
      for (int w : G.adj(sg.index(source)))
         StdOut.println("   " + sg.name(w));
   }
}
```

```
% java SymbolGraph routes.txt " "
JFK
   ORD
   ATL
   MCO
```

```
LAX
  LAS
  PHX
% java SymbolGraph movies.txt "/"
Tin Men (1987)
  DeBoy, David
  Blumenfeld, Alan
  ...
  Geppi, Cindy
  Hershey, Barbara
  ...
Bacon, Kevin
  Mystic River (2003)
  Friday the 13th (1980)
  Flatliners (1990)
  Few Good Men, A (1992)   ...
```

Этот подход, очевидно, эффективен для любых рассматриваемых нами методов обработки графов: любой клиент может использовать функцию `index()` для преобразования имени вершины в индекс, который нужен для обработки графа, и функцию `name()` для преобразования индекса вершины в имя, которое нужно в прикладном контексте.

Реализация

Полная реализация API `SymbolGraph` приведена в листинге 4.1.10 (см. также рис. 4.1.32). Она создает три структуры данных:

- таблица имен `st` с ключами типа `String` (имена вершин) и значениями типа `int` (индексы);
- массив `keys[]`, который играет роль инвертированного индекса и выдает имя вершины, связанное с каждым целочисленным индексом;
- граф `G`, в котором индексы позволяют обращаться к вершинам.

Листинг 4.1.10. Тип данных символьного графа

```
public class SymbolGraph
{
   private ST<String, Integer> st;            // строка -> индекс
   private String[] keys;                     // индекс -> строка
   private Graph G;                           // граф

   public SymbolGraph(String stream, String sp)
   {
      st = new ST<String, Integer>();
      In in = new In(stream);                 // Первый проход
      while (in.hasNextLine())                //   создает индекс:
      {
         String[] a = in.readLine().split(sp); //   он читает строки
         for (int i = 0; i < a.length; i++)    //   и связывает каждую
            if (!st.contains(a[i]))            //   отдельную строку
               st.put(a[i], st.size());        //   с индексом.
      }
```

```
    keys = new String[st.size()];           // Инвертированный индекс
    for (String name : st.keys())           //   для получения строковых ключей
      keys[st.get(name)] = name;            //   представляет собой массив.
    G = new Graph(st.size());
    in = new In(stream);                    // Второй проход
    while (in.hasNextLine())                //   создает граф:
    {
      String[] a = in.readLine().split(sp); //   он соединяет
      int v = st.get(a[0]);                 //   первую вершину
      for (int i = 1; i < a.length; i++)    //   в каждой строке
         G.addEdge(v, st.get(a[i]));        //   со всеми остальными.
    }
  }
  public boolean contains(String s)   {  return st.contains(s);  }
  public int index(String s)          {  return st.get(s);  }
  public String name(int v)           {  return keys[v];  }
  public Graph G()                    {  return G;  }
}
```

Этот клиент Graph позволяет клиентам определять графы со строковыми именами вершин, а не с целочисленными индексами. Он использует переменные экземпляров st (таблица имен для отображения имен в индексы), keys (массив для отображения индексов в имена) и G (граф с целочисленными именами вершин). Для создания этих структур данных выполняются два прохода по определению графа (каждая строка файла представляет собой строковое имя вершины и список смежных с ней вершин, разделяемых символом sp).

```
% java DegreesOfSeparation movies.txt "/" "Bacon, Kevin"
Kidman, Nicole
   Bacon, Kevin
   Few Good Men, A (1992)
   Cruise, Tom
   Days of Thunder (1990)
   Kidman, Nicole
Grant, Cary
   Bacon, Kevin
   Mystic River (2003)
   Willis, Susan
   Majestic, The (2001)
   Landau, Martin
   North by Northwest (1959)
   Grant, Cary
```

Класс SymbolGraph выполняет два прохода по данным, чтобы построить эти структуры данных — в основном потому, что для создания объекта Graph необходимо количество вершин V. В типичных реальных приложениях хранение значений V и E в файле определения графа (как в нашем конструкторе Graph в начале данного раздела) не совсем удобно; реализация SymbolGraph позволяет работать с файлами наподобие routes.txt или movies.txt, добавляя и удаляя в них элементы и не заботясь о количестве содержащихся в них различных имен.

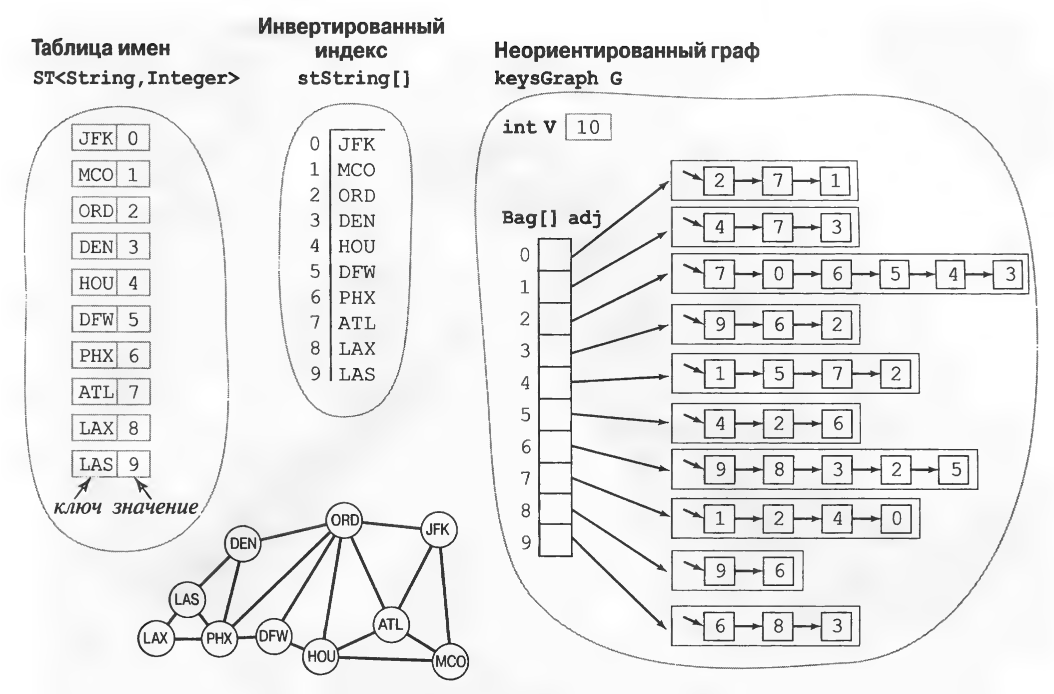


Рис. 4.1.32. Структуры данных символьного графа

Степени разделения

Одно из классических применений обработки графов — поиск степени разделения между двумя лицами в социальной сети. Чтобы было понятнее, мы объясним эту идею на примере недавно популярного развлечения — *игры Кевина Бэкона*, в которой используется знакомый нам граф с киноактерами. Кевин Бэкон — популярный актер, который снимался во многих фильмах. Каждому актеру присваивается *число Бэкона* следующим образом: самому Бэкону соответствует число 0, любой актер, который снимался в том же фильме, что и Бэкон, получает число 1, любой другой актер (кроме Бэкона), который снимался в фильмах с актерами с номером 1, обретает число 2 и т.д. Например, Мэрил Стрип присваивается число Бэкона 1, потому что она снималась с ним в фильме "Дикая река". Николь Кидман — обладательница числа Бэкона 2: с ним самим она не снималась, но снималась в фильме "Дни грома" с Томом Крузом, а Круз снимался с Кевином Бэконом в фильме "Несколько хороших парней". То есть игра сводится к тому, чтобы для любого заданного актера найти последовательность фильмов и актеров, которая приводит к Кевину Бэкону. Например, фанатик кино может знать, что Том Хэнкс снимался в "Джо против вулкана" с Ллойдом Бриджзом, который снимался в "Ровно в полдень" с Грейс Келли, которая снималась в "В случае убийства набирайте «М»" с Патриком Алленом, который снимался в "Орел приземлился" с Дональдом Сазерлендом, который снимался в фильме "Зверинец" с Кевином Бэконом. Но эта последовательность никак не поможет установить число Бэкона для Тома Хэнкса: реально оно равно 1, т.к. он снимался с Бэконом в фильме "Аполлон 13". Очевидно, что число Бэкона определяется подсчетом фильмов в *кратчайшей* такой последовательности, и трудно определить победителя в такой игре, не пользуясь компьютером.

Из клиента DegreesOfSeparation класса SymbolGraph (листинг 4.1.11) понятно, что для определения числа Бэкона для любого актера из файла movies.txt нужна программа поиска кратчайших путей, подобная BreadthFirstPaths. Эта программа принимает из командной строки исходную вершину, а затем выбирает из стандартного ввода запросы и выводит кратчайший путь из источника до вершины, указанной в запросе. В силу двудольности графа, связанного с файлом movies.txt, все пути проходят поочередно через фильмы и актеров, и выведенный путь является "доказательством", что этот путь верен (но не доказательством того, что он самый короткий — для этого надо ознакомить пользователей с утверждением Б). Клиент DegreesOfSeparation находит кратчайшие пути и в графах, которые не являются двудольными: например, он может найти способ попасть из одного аэропорта в другой из файла routes.txt с наименьшим количеством пересадок.

Алгоритм 4.1.11. Степени разделения

```
public class DegreesOfSeparation
{
   public static void main(String[] args)
   {
      SymbolGraph sg = new SymbolGraph(args[0], args[1]);
      Graph G = sg.G();
      String source = args[2];
      if (!sg.contains(source))
      {  StdOut.println(source + " отсутствует в базе данных."); return;  }

      int s = sg.index(source);
      BreadthFirstPaths bfs = new BreadthFirstPaths(G, s);

      while (!StdIn.isEmpty())
      {
         String sink = StdIn.readLine();
         if (sg.contains(sink))
         {
            int t = sg.index(sink);
            if (bfs.hasPathTo(t))
               for (int v : bfs.pathTo(t))
                  StdOut.println("   " + sg.name(v));
            else StdOut.println("Не связаны");
         }
         else StdOut.println("Нет в базе данных.");
      }
   }
}
```

Этот клиент SymbolGraph и BreadthFirstPaths находит кратчайшие пути в графах. В случае файла movies.txt он помогает играть в игру Кевина Бэкона.

```
% java DegreesOfSeparation routes.txt " " JFK
LAS
   JFK
   ORD
   PHX
   LAS
```

```
DFW
   JFK
   ORD
   DFW
% java DegreesOfSeparation movies.txt "/" "Bacon, Kevin"
Kidman, Nicole
   Bacon, Kevin
   Few Good Men, A (1992)
   Cruise, Tom
   Days of Thunder (1990)
   Kidman, Nicole
Grant, Cary
   Bacon, Kevin
   Mystic River (2003)
   Willis, Susan
   Majestic, The (2001)
   Landau, Martin
   North by Northwest (1959)
   Grant, Cary
% java DegreesOfSeparation movies.txt "/" "Animal House (1978)"
Titanic (1997)
   Animal House (1978)
   Allen, Karen (I)
   Raiders of the Lost Ark (1981)
   Taylor, Rocky (I)
   Titanic (1997)
To Catch a Thief (1955)
   Animal House (1978)
   Vernon, John (I)
   Topaz (1969)
   Hitchcock, Alfred (I)
   To Catch a Thief (1955)
```

Клиент `DegreesOfSeparation` позволяет отвечать на некоторые интересные вопросы, связанные с кинематографом. Например, можно находить степени разделения не между актерами, а между фильмами. Однако концепция разделения (или *отчуждения*) широко применяется и во многих других контекстах. Например, математики играют в эту же игру с графами, которые определяются научными статьями и соавторами по отношению к П. Эрдешу (P. Erdös) — известному математику XX века. Аналогично, похоже, что каждый житель Одессы имеет число тети Сони, равное 2 — каждый в этом городе заявляет, что знает того, кто точно был знаком с героиней песни “Семь сорок”. Для игры Ердеша необходима база данных всех математических статей, но вот с тетей Соней дела обстоят несколько сложнее. А если серьезно, то степени разделения играют важную роль в проектировании компьютеров и электронных сетей, а также в сетях, естественно возникающих во всех областях науки.

Резюме

В данном разделе мы познакомились с несколькими базовыми концепциями, которые будут углубляться в остальной части этой главы.

- Терминология, принятая при работе с графами.
- Представление графов, которое позволяет обрабатывать огромные разреженные графы.
- Классический способ реализации алгоритмов: разработка клиентов, которые выполняют предварительную обработку графа в конструкторе и строят структуры данных, которые могут эффективно отвечать на клиентские запросы о графе.
- Поиск в глубину и поиск в ширину.
- Класс, позволяющий использовать символьные имена вершин.

В табл. 4.1.4 приведена сводка реализаций изученных нами алгоритмов на графах. Эти алгоритмы представляют собой хорошее введение в обработку графов, т.к. варианты их кода постоянно будут встречаться нам при рассмотрении более сложных видов графов и приложений и (следовательно) более сложных задач обработки графов. Эти же вопросы о связности и путях между вершинами становятся значительно сложнее при навешивании на ребра графа направлений и весов, но для ответов на них такие подходы могут служить начальной точкой.

Таблица 4.1.4. Задачи обработки (неориентированных) графов, рассмотренные в данном разделе

Задача	Решение	Где
Связность с одним источником	`DepthFirstSearch`	Листинг 4.1.3
Пути из одного источника	`DepthFirstPaths`	Листинг 4.1.5
Кратчайшие пути из одного источника	`BreadthFirstPaths`	Листинг 4.1.6
Связность	`CC`	Листинг 4.1.8
Обнаружение циклов	`Cycle`	Таблица 4.1.3
Двухцветность (двудольность)	`TwoColor`	Таблица 4.1.3

Вопросы и ответы

Вопрос. Почему бы не собрать все алгоритмы в файл `Graph.java`?

Ответ. Да, можно просто добавить в базовое определение АТД `Graph` методы запросов (и необходимые для этого приватные поля и методы). Такой способ имеет некоторые преимущества абстракции данных — но он имеет и серьезные недостатки, т.к. мир обработки графов растет значительно быстрее, чем базовые структуры данных из раздела 1.3. Основные недостатки перечислены ниже.

- Операций обработки графов, достойных реализации, существует гораздо больше, чем операций, которые можно аккуратно реализовать в одном API.
- Для простых задач обработки графов придется использовать тот же API, что и для сложных задач.

- Один метод сможет обращаться к полям, предназначенным для другого метода, а это противоречит принципу инкапсуляции, которому мы стараемся следовать.

Эта ситуация не такая уж необычная, и подобные API часто называются *широкими* интерфейсами (см. раздел 1.2). В главе, насыщенной алгоритмами обработки графов, такой API будет действительно широким.

Вопрос. А что, в реализации `SymbolGraph` действительно необходимы два прохода?

Ответ. Нет. За счет увеличения трудоемкости до $\lg N$ метод `adj()` может работать непосредственно с классом `ST`, а не `Bag`. Такой подход применяется в нашей книге *An Introduction to Programming in Java: An Interdisciplinary Approach*.

Упражнения

4.1.1. Каково максимальное количество ребер в графе с V вершинами без параллельных ребер? Каково минимальное количество ребер в графе с V вершинами, среди которых нет изолированных?

4.1.2. Изобразите в стиле рисунков данного раздела (см. рис. 4.1.10) списки смежности, построенные конструктором входного потока класса `Graph` для файла `tinyGex2.txt`, приведенного на рис. 4.1.33.

4.1.3. Напишите конструктор копирования для класса `Graph`, который принимает в качестве параметра граф `G` и создает и инициализирует новую копию этого графа. Любые изменения, выполненные клиентом с `G`, не должны влиять на новый граф.

4.1.4. Добавьте в класс `Graph` метод `hasEdge()`, который принимает два целочисленных аргумента `v` и `w` и возвращает `true`, если в графе имеется ребро `v-w`, и `false` в противном случае.

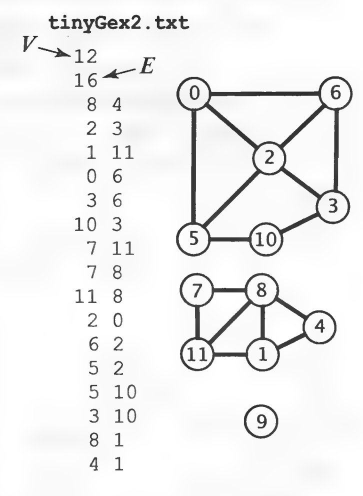


Рис. 4.1.33. *Граф для упражнения 4.1.2*

4.1.5. Исправьте класс `Graph`, чтобы в нем не допускались параллельные ребра и петли.

4.1.6. Пусть имеется граф с четырьмя вершинами и ребрами `0-1`, `1-2`, `2-3` и `3-0`. Приведите содержимое массива списков смежности, который *невозможно* построить с помощью вызовов `addEdge()` для этих ребер *в любом порядке*.

4.1.7. Напишите клиент тестирования для класса `Graph`, который читает граф из входного потока с именем, вводимым в качестве аргумента из командной строки, а затем выводит его с помощью метода `toString()`.

4.1.8. Разработайте реализацию для API `Search` (см. рис. 4.1.12), в которой используется реализация `UF`, как описано в тексте.

4.1.9. Приведите в стиле рис. 4.1.18 подробную трассировку вызова `dfs(0)` для графа, построенного конструктором входного потока `Graph` для файла `tinyGex2.txt` (см. упражнение 4.1.2). Начертите также дерево, представленное массивом `edgeTo[]`.

4.1.10. Докажите, что в каждом связном графе имеется вершина, удаление которой (и всех смежных с ней ребер) не разъединит граф. Напишите метод на основе ПвГ, который находит такую вершину. *Совет*: рассматривайте вершины, у которых помечены все смежные с ней вершины.

4.1.11. Нарисуйте дерево, представленное массивом `edgeTo[]`, после вызова `bfs(G, 0)` в алгоритме 4.2 для графа, построенного конструктором входного потока класса `Graph` для файла `tinyGex2.txt` (см. упражнение 4.1.2).

4.1.12. Что говорит дерево ПвШ о расстоянии от `v` до `w`, если ни одна из этих вершин не является корнем?

4.1.13. Добавьте в API `BreadthFirstPaths` метод `distTo()` (и его реализацию), который возвращает количество ребер на кратчайшем пути от источника до указанной вершины. Запрос `distTo()` должен отрабатываться за константное время.

4.1.14. Допустим, что при выполнении поиска в ширину используется не очередь, а стек. Будет ли он все равно вычислять кратчайшие пути?

4.1.15. Измените конструктор входного потока для класса `Graph`, чтобы принимать списки смежности из стандартного ввода (аналогично `SymbolGraph`), вроде файла `tinyGadj.txt`, приведенного на рис. 4.1.34. После количества вершин и ребер каждая строка содержит вершину и список смежных с ней вершин.

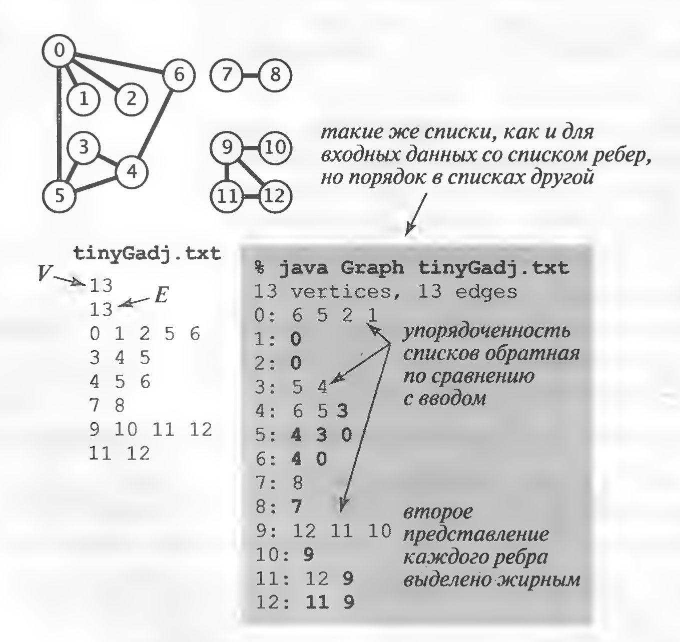


Рис. 4.1.34. Граф для упражнения 4.1.15

4.1.16. *Эксцентриситетом* вершины `v` является длина кратчайшего пути от вершины, которая находится на наибольшем расстоянии от `v`. *Диаметр* графа — это максимальный эксцентриситет его вершин. *Радиус* графа — наименьший эксцентриситет его вершин. *Центральная* вершина — это вершина, эксцентриситет которой равен радиусу графа. Реализуйте API, приведенный на рис. 4.1.35.

public class **GraphProperties**		
	GraphProperties(Graph G)	*конструктор (генерирует исключение, если* G *не связный)*
int	eccentricity(int v)	*эксцентриситет* v
int	diameter()	*диаметр* G
int	radius()	*радиус* G
int	center()	*центр* G

***Рис. 4.1.35.** API к упражнению 4.1.16*

4.1.17. *Обхват* графа — это длина его кратчайшего цикла. Если граф ацикличен, то его обхват равен бесконечности. Добавьте в класс GraphProperties метод girth(), который возвращает обхват графа. *Совет*: выполните ПвШ для каждой вершины. Кратчайший цикл, содержащий вершину s — это кратчайший путь из s до некоторой вершины v плюс ребро от v назад в s.

4.1.18. Приведите в стиле рис. 4.1.28 подробную трассировку работы класса CC при нахождении связных компонентов в графе, построенном конструктором входного потока класса Graph для файла tinyGex2.txt (см. упражнение 4.1.2).

4.1.19. Приведите в стиле рисунков из данного раздела подробную трассировку работы класса Cycle при нахождении цикла в графе, построенном конструктором входного потока класса Graph для файла tinyGex2.txt (см. упражнение 4.1.2). Каков порядок роста времени работы конструктора Cycle в худшем случае?

4.1.20. Приведите в стиле рисунков из данного раздела подробную трассировку работы класса TwoColor при нахождении цикла в графе, построенном конструктором входного потока класса Graph для файла tinyGex2.txt (см. упражнение 4.1.2). Каков порядок роста времени работы конструктора TwoColor в худшем случае?

4.1.21. Выполните программу SymbolGraph с файлом movies.txt, чтобы найти число Бэкона для номинантов на премию Оскар этого года.

4.1.22. Напишите программу BaconHistogram, которая выводит гистограмму чисел Бэкона с подсчетом количества актеров из файла movies.txt, для которых число Бэкона равно 0, 1, 2, 3, ... Предусмотрите отдельную категорию для актеров с бесконечным числом Бэкона (не связанных с ним).

4.1.23. Вычислите количество связных компонентов в файле movies.txt, размер наибольшего компонента и количество компонентов размером менее 10. Найдите эксцентриситет, диаметр, радиус, центр и обхват наибольшего компонента (см. упражнения 4.1.16 и 4.1.17). Содержит ли этот компонент Кевина Бэкона?

4.1.24. Измените код клиента DegreesOfSeparation, чтобы он принимал значение y типа int в качестве аргумента командной строки и игнорировал фильмы старше y лет.

4.1.25. Напишите клиент SymbolGraph, аналогичный DegreesOfSeparation, который для нахождения путей, соединяющих двух актеров, использует поиск не в ширину, а *в глубину*, и выводит результат вроде приведенного в листинге 4.1.12.

Листинг 4.1.12. Пример выходных данных для упражнения 4.1.25

```
% java DegreesOfSeparationDFS movies.txt
Source: Bacon, Kevin
Query:  Kidman, Nicole
   Bacon, Kevin
   Mystic River (2003)
   O'Hara, Jenny
   Matchstick Men (2003)
   Grant, Beth
   ... [123 фильма ] (!)
   Law, Jude
   Sky Captain... (2004)
   Jolie, Angelina
   Playing by Heart (1998)
   Anderson, Gillian (I)
   Cock and Bull Story, A (2005)
   Henderson, Shirley (I)
   24 Hour Party People (2002)
   Eccleston, Christopher
   Gone in Sixty Seconds (2000)
   Balahoutis, Alexandra
   Days of Thunder (1990)
   Kidman, Nicole
```

4.1.26. Определите объем памяти, необходимый классу Graph для представления графа с V вершинами и E ребрами. Используйте модель требований к памяти из раздела 1.4.

4.1.27. Два графа называются *изоморфными*, если можно переименовать вершины одного из них, чтобы он стал идентичен другому. Нарисуйте все не изоморфные графы с двумя, тремя, четырьмя и пятью вершинами.

4.1.28. Измените код Cycle так, чтобы он работал даже при наличии в графе петель и параллельных ребер.

Творческие задачи

4.1.29. *Эйлеровы и гамильтоновы циклы.* Даны графы, определенные следующими четырьмя множествами ребер:

```
0-1 0-2 0-3 1-3 1-4 2-5 2-9 3-6 4-7 4-8 5-8 5-9 6-7 6-9 7-8
0-1 0-2 0-3 1-3 0-3 2-5 5-6 3-6 4-7 4-8 5-8 5-9 6-7 6-9 8-8
0-1 1-2 1-3 0-3 0-4 2-5 2-9 3-6 4-7 4-8 5-8 5-9 6-7 6-9 7-8
4-1 7-9 6-2 7-3 5-0 0-2 0-8 1-6 3-9 6-3 2-8 1-5 9-8 4-5 4-7
```

В каких из этих графов имеются эйлеровы циклы (циклы, проходящие по каждому ребру в точности один раз)? В каких из этих графов имеются гамильтоновы циклы (циклы, проходящие через каждую вершину в точности один раз)?

4.1.30. *Перечисление графов.* Сколько существует различных неориентированных графов с V вершинами и E ребрами (без параллельных ребер)?

4.1.31. *Обнаружение параллельных ребер.* Разработайте алгоритм с линейным временем выполнения для подсчета параллельных ребер в графе.

4.1.32. *Нечетные циклы.* Докажите, что граф является двухцветным (двудольным) тогда и только тогда, когда он не содержит циклов нечетной длины.

4.1.33. *Символьный граф.* Реализуйте однопроходный API `SymbolGraph` (не должен быть клиентом класса `Graph`). В вашей реализации допустимы дополнительные затраты порядка $\log V$ для выполнения операций с графами, связанных с поиском в таблице имен.

4.1.34. *Двусвязность.* Граф называется *двусвязным*, если каждая пара его вершин связана двумя отдельными путями. *Точкой сочленения* в связном графе называется вершина, которая разъединит граф, если удалить ее и смежные с ней ребра. Докажите, что любой граф без точек сочленения является двусвязным. *Совет*: рассмотрите пару вершин `s` и `t` и соединяющий их путь. Используйте тот факт, что ни одна вершина в графе не является точкой сочленения, для построения двух отдельных путей, соединяющих `s` и `t`.

4.1.35. *Реберная связность. Мостом* в графе называется такое ребро, что его удаление приведет к разбиению связного графа на два не связанных подграфа. Граф, не содержащий мостов, называется *реберно связным*. Разработайте тип данных, который позволяет с помощью ПвГ определить, является ли заданный граф реберно связным.

4.1.36. *Евклидовы графы.* Напишите и реализуйте API `EuclideanGraph` для графов, вершины которых представляют собой точки на плоскости с указанными координатами. Включите в API метод `show()` для вычерчивания такого графа с помощью `StdDraw`.

4.1.37. *Обработка изображений.* Реализуйте операцию *заливки* в графе, который неявно определяется соединением смежных точек изображения, имеющих один цвет.

Эксперименты

4.1.38. *Случайные графы.* Напишите программу `ErdosRenyiGraph`, которая принимает из командной строки целочисленные значения V и E и строит граф, генерируя E случайных пар целых чисел от 0 до $V-1$. *Примечание*: такой генератор может создавать петли и параллельные ребра.

4.1.39. *Случайные простые графы.* Напишите программу `RandomSimpleGraph`, которая принимает из командной строки целочисленные значения V и E и строит с равной вероятностью один из возможных *простых* графов с V вершинами и E ребрами.

4.1.40. *Случайные разреженные графы.* Напишите программу `RandomSparseGraph`, которая генерирует случайные разреженные графы для подобранных наборов значений V и E, чтобы можно было выполнять осмысленные эмпирические тесты на графах на основе модели Ердеша-Реньи.

4.1.41. *Случайные евклидовы графы.* Напишите клиент `RandomEuclideanGraph` класса `EuclideanGraph` (см. упражнение 4.1.36), который создает случайные графы, генерируя V случайных точек на плоскости, а затем соединяя каждую точку со всеми точками из круга радиусом d с центром в этой точке. *Примечание*: такой граф почти наверняка будет связным, если d больше порогового значения $\lg V/V$, и почти наверняка будет не связным, если d меньше этого значения.

4.1.42. *Случайные графы на решетке.* Напишите клиент RandomGridGraph класса EuclideanGraph, который создает случайные графы, соединяя с соседями вершины, расположенные на решетке $V \times V$ (см. упражнение 1.5.18), после чего добавляет еще R случайных ребер. Для больших R ужмите решетку, чтобы общее количество ребер имело порядок V. Добавьте правило, что дополнительные ребра соединяют вершину s с вершиной t с вероятностью, пропорциональной расстоянию между s и t.

4.1.43. *Реальные графы.* Найдите в Интернете большой взвешенный граф — например, карту с расстояниями, телефонные соединения с их стоимостями, или таблицу стоимостей авиаперелетов. Напишите программу RandomRealGraph, которая строит граф, выбирая случайным образом из исходных данных V вершин и E ребер, индуцированных этими вершинами.

4.1.44. *Случайный интервальный граф.* Пусть имеется набор V интервалов на вещественной оси (пары вещественных чисел). Такой набор определяет *интервальный граф*, в котором вершины соответствуют интервалам, а ребра между вершинами — пересечениям этих интервалов (когда они имеют общие точки). Напишите программу, которая генерирует на единичном отрезке V случайных интервалов длиной d, а затем строит соответствующий интервальный граф. *Совет*: используйте ПвШ.

4.1.45. *Случайные транспортные графы.* Транспортную систему можно (в частности) задать набором последовательностей вершин, где каждая такая последовательность определяет путь, соединяющий вершины. Например, последовательность 0-9-3-2 определяет ребра 0-9, 9-3 и 3-2. Напишите клиент RandomTransportation класса EuclideanGraph, который строит граф из входного файла, содержащего в каждой строке одну последовательность символьных имен вершин. Создайте входные данные, на основании которых программа построит граф, соответствующий системе парижского метро.

Тестирование всех алгоритмов со всеми возможными параметрами на всех моделях графов выполнить нереально. Для каждой из приведенных ниже задач напишите клиент, решающий эту задачу, а затем выберите один из описанных выше генераторов, чтобы проводить эксперименты для данной модели графов. Планируйте эксперименты обдуманно, возможно, на основе результатов предыдущих экспериментов. Напишите краткий анализ полученных результатов и выводы из этих результатов.

4.1.46. *Длины путей в ПвГ.* Экспериментально определите вероятность, что программа DepthFirstPaths найдет путь между двумя случайно выбранными вершинами, и вычислите среднюю длину найденных путей для различных моделей графов.

4.1.47. *Длины путей в ПвШ.* Экспериментально определите вероятность, что программа BreadthFirstPaths найдет путь между двумя случайно выбранными вершинами, и вычислите среднюю длину найденных путей для различных моделей графов.

4.1.48. *Связные компоненты.* Экспериментально определите распределение количеств компонентов в случайных графах различных видов, сгенерировав большое количество графов. Нарисуйте гистограмму.

4.1.49. *Двухцветность.* Большинство графов не являются двухцветными, и ПвГ позволяет быстро это выяснить. Экспериментально определите количество ребер, проверенных программой TwoColor для различных моделей графов.

4.2. Ориентированные графы

В *ориентированных графах* ребра однонаправленные: пара вершин, определяющая каждое ребро, представляет собой упорядоченную пару, которая задает одностороннюю смежность. Многие приложения (например, графы, представляющие веб, ограничения в расписаниях, телефонные звонки) естественно описываются как раз ориентированными графами (табл. 4.2.1). Односторонние ограничения понятны, легко реализуются и выглядят вполне безобидно; однако они приводят к комбинаторным структурам, которые существенно влияют на алгоритмы и делают работу с ориентированными графами весьма непохожей на работу с неориентированными графами. В данном разделе мы рассмотрим классические алгоритмы для исследования и обработки ориентированных графов.

Таблица 4.2.1. Типичные применения орграфов

Применение	Вершина	Ребро
Пищевая цепочка	Виды	Хищник-жертва
Интернет-контент	Страница	Гиперссылка
Программа	Модуль	Внешняя ссылка
Сотовая связь	Телефон	Звонок
Научные публикации	Статья	Цитата
Финансы	Счет	Транзакция
Интернет	Компьютер	Подключение

Термины

Определения для ориентированных графов практически совпадают с определениями для неориентированных графов (да и некоторые алгоритмы и программы тоже), но все же их лучше привести здесь отдельно. Из небольших различий в словах, учитывающих направленность ребер, вытекают структурные свойства, которые и будут основной темой данного раздела.

Определение. *Ориентированный граф* (или *орграф*) — это множество *вершин* и коллекция *ориентированных ребер*. Каждое ориентированное ребро соединяет упорядоченную пару вершин.

Мы говорим, что ориентированное ребро *исходит* из первой вершины пары и *входит* во вторую вершину. *Полустепень исхода* вершины в орграфе равна количеству ребер, исходящих *из* нее; *полустепень захода* равна количеству ребер, входящих *в* нее. Когда мы говорим о ребрах в орграфе и различие понятно из контекста, мы не будем употреблять слово *ориентированный*. Первая вершина в ориентированном ребре называется ее *началом*, а вторая — ее *концом*. На рисунках ориентированные ребра изображаются стрелками, указывающими от начала к концу (рис. 4.2.1). Мы будем использовать обозначение v->w для ребра в орграфе, направленного из v в w. Как и в случае неориентированных графов, наш код может обрабатывать параллельные ребра и петли, но в примерах их не будет, и в тексте мы обычно не будем упоминать о них.

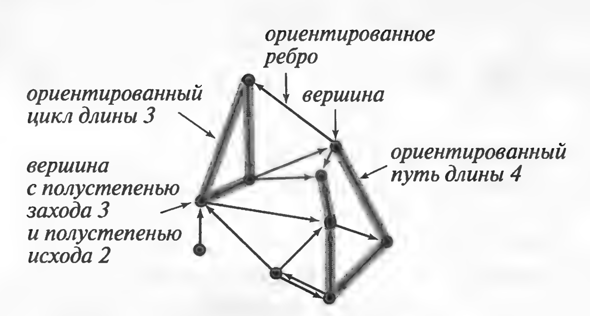


Рис. 4.2.1. *Анатомия орграфа*

Не считая аномалий, две вершины могут быть связаны в орграфе четырьмя различными способами: ребро отсутствует, ребро v->w из v в w, ребро w->v из w в v, или два ребра v->w и w->v, связывающие их в обоих направлениях.

Определение. *Ориентированный путь* в орграфе — это последовательность вершин, в которой существует (ориентированное) ребро, указывающее из каждой вершины на следующую вершину этой последовательности. *Ориентированный цикл* — это ориентированный путь, содержащий, по крайней мере, одно ребро, в котором первая и последняя вершины совпадают. *Простой цикл* — это цикл без повторяющихся ребер или вершин (кроме обязательного совпадения первой и последней вершин). *Длина* пути или цикла равна количеству ребер в нем.

Как и в случае неориентированных графов, мы считаем, что ориентированные пути являются простыми — кроме тех случаев, когда мы специально ослабим это требование по отношению к отдельным повторяющимся вершинам (как в определении ориентированного цикла) или к ориентированным путям *общего вида*. Мы будем говорить, что вершина w *достижима* из вершины v, если существует ориентированный путь из v в w. Кроме того, мы будем считать, что каждая вершина достижима из себя самой. Кроме этого случая, из факта достижимости w из v в орграфе нельзя сделать вывод, достижима ли v из w. Как мы вскоре увидим, это очевидное различие очень важно.

Для понимания алгоритмов из данного раздела необходимо уяснить различие между достижимостью в орграфах и связностью в неориентированных графах. Такое уяснение не так просто, как кажется. Например, нетрудно с одного взгляда сказать, связаны ли две вершины в небольшом неориентированном графе, но ориентированный путь в орграфе обнаружить сложнее, как показано на рис. 4.2.2. Обработка орграфов похожа на поездку по городу, в котором все улицы с односторонним движением, а направления этого движения никак не упорядочены. Перемещение из одного места в другое может стать в такой ситуации сложной задачей. Но, несмотря на эту сложность, стандартная структура данных, которую мы будем использовать для представления орграфов, *проще* соответствующего представления для неориентированных графов!

Тип данных орграфа

API, показанный на рис. 4.2.3, и класс Digraph, приведенный в листинге 4.2.1, почти совпадают с классом Graph (см. листинг 4.1.1).

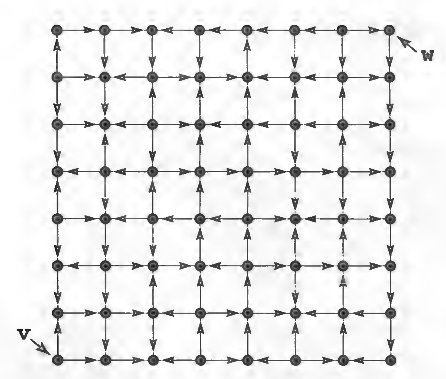


Рис. 4.2.2. Достижима ли вершина w *из* v *в этом орграфе?*

public class **Digraph**		
	Digraph(int V)	*создание орграфа с* V *вершинами без ребер*
	Digraph(In in)	*чтение орграфа из входного потока* in
int	V()	*количество вершин в орграфе*
int	E()	*количество ребер в орграфе*
void	addEdge(int v, int w)	*добавление в орграф ребра* v->w
Iterable<Integer>	adj(int v)	*вершины, связанные с* v *ребрами, направленными из* v
Digraph	reverse()	*обращение орграфа*
String	toString()	*строковое представление*

Рис. 4.2.3. API для орграфа

Листинг 4.2.1. Тип данных ориентированного графа (орграфа)

```
public class Digraph
{
  private final int V;
  private int E;
  private Bag<Integer>[] adj;

  public Digraph(int V)
  {
    this.V = V;
    this.E = 0;
    adj = (Bag<Integer>[]) new Bag[V];
    for (int v = 0; v < V; v++)
      adj[v] = new Bag<Integer>();
  }
  public int V()  {  return V;  }
  public int E()  {  return E;  }
  public void addEdge(int v, int w)
  {
    adj[v].add(w);
    E++;
  }
```

```
  public Iterable<Integer> adj(int v)
  {  return adj[v];  }

  public Digraph reverse()
  {
     Digraph R = new Digraph(V);
     for (int v = 0; v < V; v++)
        for (int w : adj(v))
           R.addEdge(w, v);
     return R;
  }
}
```

Этот тип данных `Digraph` похож на тип `Graph` (листинг 4.1.1), но метод `addEdge()` вызывает `add()` только один раз, и имеется дополнительный метод `reverse()`, который возвращает копию графа с перевернутыми ребрами. Этот код легко получить из соответствующего кода типа `Graph`, поэтому в нем опущены метод `toString()` (см. табл. 4.1.1) и конструктор из входного потока (листинг 4.1.1).

Представление

Мы будем использовать представление списками смежности, где ребро v->w представлено узлом списка, содержащим w в связном списке, который соответствует вершине v. Это представление почти совпадает с представлением для неориентированных графов, но оно логичнее, поскольку каждое ребро присутствует в нем только один раз, как показано на рис. 4.2.4.

Формат входных данных

Код конструктора, который принимает орграф из входного потока, идентичен соответствующему конструктору из класса `Graph`: формат входных данных тот же, но все ребра считаются ориентированными. В формате списков ребер пара v w интерпретируется как ребро v->w.

Обращение орграфа

В API `Digraph` добавлен метод `reverse()`, который возвращает копию орграфа, в которой направления всех ребер изменены на обратные. Этот метод иногда нужен в обработке орграфов, т.к. он позволяет клиентам находить ребер, направленные *в* каждую вершину, в то время как метод `adj()` выдает только вершины, связанные ребрами, которые направлены *из* вершин.

Символьные имена

В приложениях обработки орграфов нетрудно дать возможность клиентам использовать символьные имена. Для реализации класса `SymbolDigraph`, аналогичного `SymbolGraph` на рис. 4.1.10, замените везде `Graph` на `Digraph`.

Не пожалейте времени и внимательно изучите разницу, сравнив листинг 4.2.1 и рис. 4.2.4 с соответствующими им листингом 4.1.1 и рис. 4.1.10. В представлении списками смежности для неориентированного графа мы знаем, что если v находится в списке вершины w, то и w находится в списке вершины v; однако в представлении списками смежности для орграфов такой симметрии нет. Эта разница имеет очень серьезные последствия в обработке орграфов.

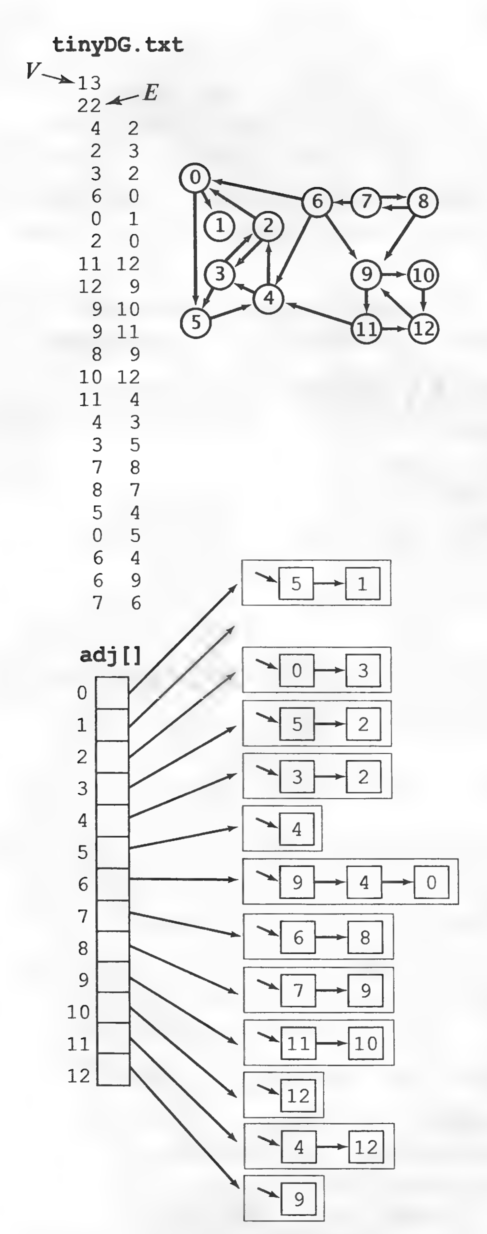


Рис. 4.2.4. Формат входных данных орграфа и представление списками смежности

Достижимость в орграфах

Наш первый алгоритм обработки неориентированных графов (DepthFirstSearch в листинге 4.1.3) решал задачу связности с одним источником и позволял клиентам определить, какие вершины связаны с указанной исходной вершиной. *Идентичный код*, с заменой Graph на Digraph, решает аналогичную задачу для орграфов.

> ***Достижимость из одного источника.*** Для заданного орграфа и исходной вершины s нужны ответы на запросы вида *Существует ли ориентированный путь из* s *в указанную вершину* v?

Код DirectedDFS в листинге 4.2.2 представляет собой лишь небольшую модификацию кода DepthFirstSearch и реализует API, приведенный на рис. 4.2.5.

Листинг 4.2.2. Алгоритм 4.4. Достижимость в орграфах

```
public class DirectedDFS
{
   private boolean[] marked;

   public DirectedDFS(Digraph G, int s)
   {
      marked = new boolean[G.V()];
      dfs(G, s);
   }

   public DirectedDFS(Digraph G, Iterable<Integer> sources)
   {
      marked = new boolean[G.V()];
      for (int s : sources)
         if (!marked[s]) dfs(G, s);
   }

   private void dfs(Digraph G, int v)
   {
      marked[v] = true;
      for (int w : G.adj(v))
         if (!marked[w]) dfs(G, w);
   }

   public boolean marked(int v)
   {  return marked[v];  }

   public static void main(String[] args)
   {
      Digraph G = new Digraph(new In(args[0]));
      Bag<Integer> sources = new Bag<Integer>();
      for (int i = 1; i < args.length; i++)
          sources.add(Integer.parseInt(args[i]));
      DirectedDFS reachable = new DirectedDFS(G, sources);
      for (int v = 0; v < G.V(); v++)
         if (reachable.marked(v)) StdOut.print(v + " ");
      StdOut.println();
   }
}
```

Эта реализация поиска в глубину позволяет клиентам проверять достижимость вершин из указанной вершины или указанного множества вершин.

```
% java DirectedDFS tinyDG.txt 1
1
% java DirectedDFS tinyDG.txt 2
0 1 2 3 4 5
% java DirectedDFS tinyDG.txt 1 2 6
0 1 2 3 4 5 6 9 10 11 12
```

public class **DirectedDFS**		
	`DirectedDFS(Digraph G, int s)`	*поиск в G вершин, достижимых из источника* `s`
	`DirectedDFS(Digraph G, Iterable<Integer> sources)`	*поиск в G вершин, достижимых из источников* `sources`
`boolean`	`marked(int v)`	*достижима ли вершина* `v`*?*

Рис. 4.2.5. *API для достижимости в орграфах*

После добавления второго конструктора, который принимает список вершин, этот API позволяет клиентам решать более обобщенную задачу.

Достижимость из нескольких источников. Для заданного орграфа и *множества* исходных вершин нужны ответы на запросы вида *Существует ли ориентированный путь из* любой *вершины множества в указанную вершину v?*

Эта задача возникает при решении классической задачи обработки строк, которая будет рассмотрена в разделе 5.4.

Для решения этих задач в коде `DirectedDFS` применяется наша стандартная парадигма обработки графов и стандартный поиск в глубину. Он вызывает для каждой вершины рекурсивный метод `dfs()`, который помечает каждую встреченную вершину.

Утверждение Г. Поиск в глубину помечает в орграфе все вершины, достижимые из указанного множества исходных вершин, за время, пропорциональное сумме полустепеней исхода всех помеченных вершин.

Доказательство. Аналогично доказательству утверждения А из раздела 4.1.

Трассировка работы этого алгоритма на нашем демонстрационном орграфе приведена на рис. 4.2.6. Эта трассировка несколько проще соответствующей трассировки для неориентированных графов, т.к. ПвГ, по сути, является алгоритмом обработки орграфов с одним представлением каждого ребра. Анализ этой трассировки может существенно помочь в понимании поиска в глубину на орграфах.

Сборка мусора по принципу маркировки и удаления

Важное применение достижимости из нескольких источников можно наблюдать в типичных системах управления памятью, в том числе и во многих реализациях Java. Орграф, в котором каждая вершина представляет объект, а каждое ребро — ссылку на объект, является хорошей моделью для использования памяти работающей Java-программой (рис. 4.2.7).

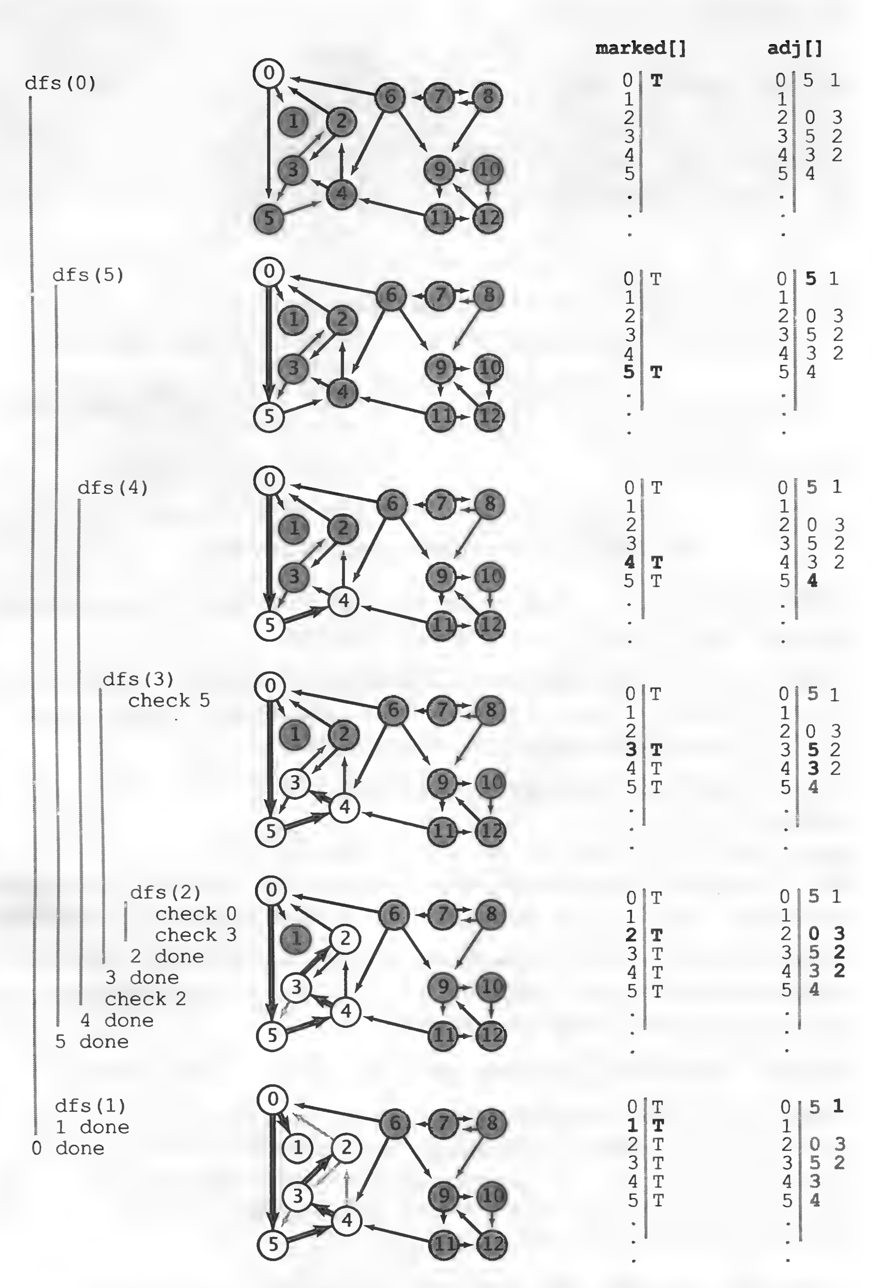


Рис. 4.2.6. Трассировка поиска в глубину для нахождения вершин, достижимых из вершины 0 в орграфе

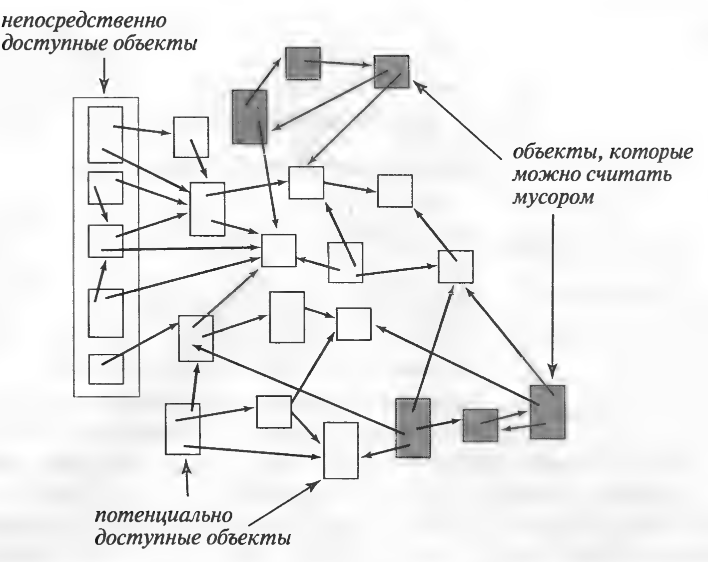


Рис. 4.2.7. Принцип сборки мусора

В любой момент выполнения программы некоторое множество объектов достижимо непосредственно, а любой объект, не достижимый из этого множества объектов, можно возвратить в пул доступной памяти. Стратегия маркировки и удаления при сборке мусора использует один бит в каждом объекте; она периодически *помечает* множество потенциально доступных объектов с помощью алгоритма достижимости в орграфе наподобие `DirectedDFS` и просматривает все объекты, *удаляя* не помеченные.

Поиск путей в орграфах

Алгоритмы `DepthFirstPaths` (алгоритм 4.1) и `BreadthFirstPaths` (алгоритм 4.2) являются также фундаментальными алгоритмами обработки орграфов. Идентичные API и код (с заменой `Graph` на `Graph`) эффективно решают следующие задачи.

Ориентированные пути из одного источника. Для заданного орграфа и исходной вершины `s` нужны ответы на запросы вида *Существует ли ориентированный путь из* `s` *в указанную вершину* `v`*?* Если да, то нужно найти такой путь.

Кратчайшие ориентированные пути из одного источника. Для заданного орграфа и исходной вершины `s` нужны ответы на запросы вида *Существует ли ориентированный путь из* `s` *в указанную вершину* `v`*?* Если да, то нужно найти *кратчайший* такой путь (с минимальным количеством ребер).

На сайте книги и в упражнениях в конце данного раздела эти решения называются `DepthFirstDirectedPaths` и `BreadthFirstDirectedPaths` соответственно.

Циклы и ориентированные ациклические графы

В приложениях, в которых обрабатываются орграфы, очень важны ориентированные циклы. Поиск ориентированного пути в типичном орграфе может оказаться весьма сложной задачей, если не использовать компьютер — взгляните, например, на рис. 4.2.8. В принципе, орграф может содержать очень много циклов, но на практике важны лишь несколько из них, или просто нужно узнать, существуют ли такие циклы.

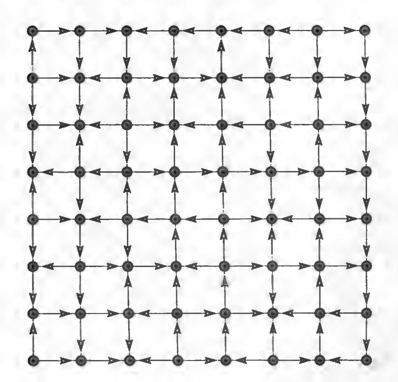


Рис. 4.2.8. Имеется ли в этом орграфе ориентированный цикл?

Чтобы подчеркнуть важность изучения ориентированных циклов в обработке орграфов, мы рассмотрим в качестве примера следующее характерное приложение, где модели орграфов используются непосредственно.

Задачи составления расписания

Широко применяемая модель решения задач — упорядочение моментов завершения множества заданий с учетом множества ограничений на то, когда и как должны выполняться задания. Ограничения могут содержать функции от необходимого времени или других ресурсов, потребляемых заданиями. Наиболее важным видом ограничений являются *ограничения предшествования*, которые указывают, что некоторые задания должны быть выполнены перед другими заданиями. Различные виды дополнительных ограничений приводят ко многим различным видам задач составления расписания, имеющим различную сложность. Изучены уже буквально тысячи различных задач, и для многих из них исследователи все еще стараются найти лучшие алгоритмы. В качестве примера мы рассмотрим преподавание предметов студентам колледжа с учетом ограничений, что некоторые курсы необходимы для понимания других (рис. 4.2.9).

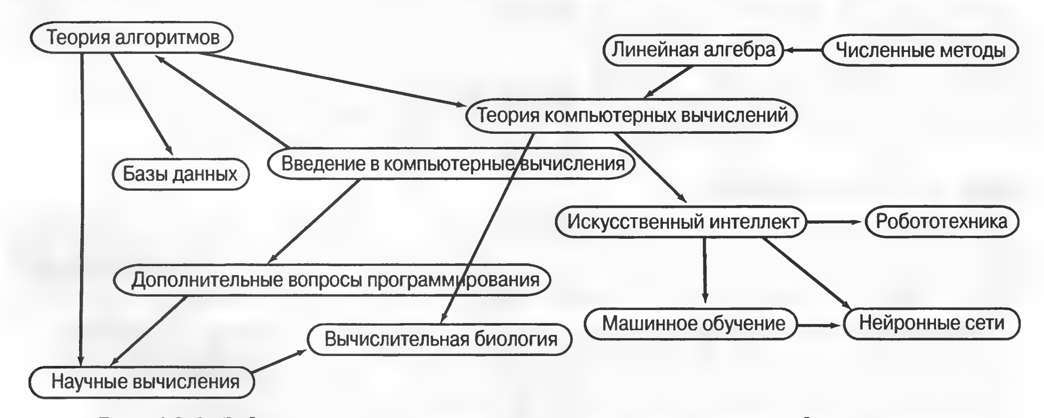


Рис. 4.2.9. Задача составления расписания с ограничениями предшествования

Если считать, что студент в любой момент времени может изучать только один курс, то возникает следующая задача.

> ***Планирование с ограничениями предшествования.*** Имеется множество заданий, которые нужно выполнить, и ограничения предшествования, которые указывают, что некоторые задания должны быть выполнены, прежде чем можно будет начать выполнение других заданий. Как можно запланировать эти задания, чтобы выполнить их, не нарушая ограничений?

Любую такую задачу удобно представить в виде орграфа, где вершины соответствуют заданиям, а ориентированные ребра — ограничениям предшествования. Для простоты мы переведем наш пример в стандартный вид с вершинами, помеченными целыми числами, как показано на рис. 4.2.10. В орграфах планирование с ограничениями предшествования сводится к следующей фундаментальной задаче.

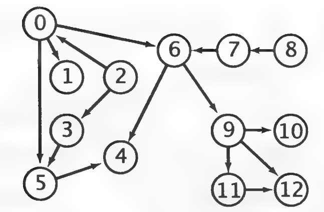


Рис. 4.2.10. Стандартная модель орграфа

Топологическая сортировка. В заданном орграфе нужно упорядочить вершины так, чтобы все его ориентированные ребра были направлены из более ранних вершин в более поздние (или сообщить, что это невозможно).

Топологическое упорядочение для нашего примера показано на рис. 4.2.11. Все ребра направлены вниз, т.е. это очевидно является решением задачи планирования с ограничениями предшествования для нашего орграфа: студент может изучать курсы с учетом необходимости одних курсов для понимания других. Это приложение довольно типично, а некоторые другие характерные примеры приведены в табл. 4.2.2.

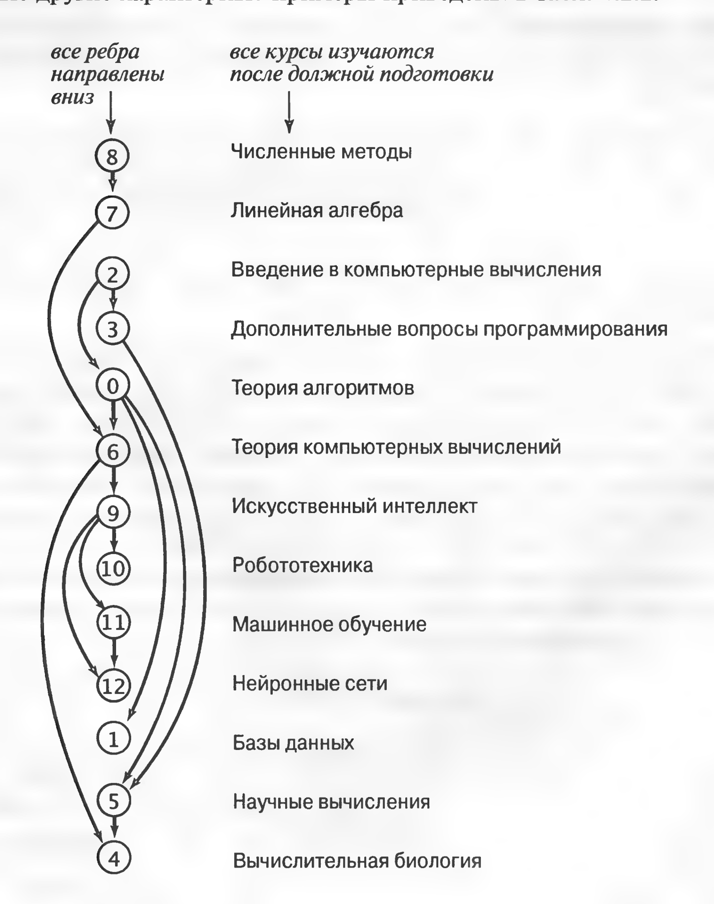


Рис. 4.2.11. Топологическая упорядоченность

Таблица 4.2.2. Типичные применения топологической сортировки

Область	Вершина	Ребро
Планирование работ	Работа	Ограничение предшествования
Преподавание предметов	Курс	Предварительный курс
Наследование	Java-класс	Расширение
Электронные таблицы	Ячейка	Формула
Символьные ссылки	Имя файла	Ссылка

Циклы в орграфах

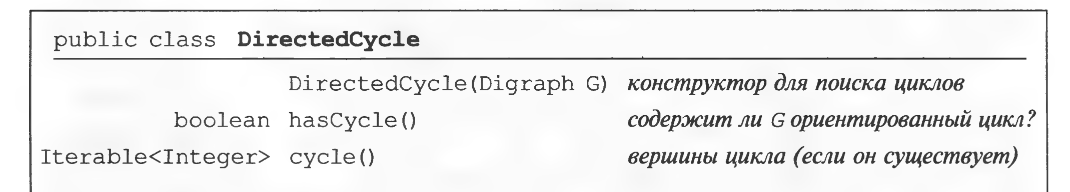
Если задание x должно быть выполнено перед заданием y, задание y перед заданием z, а задание z — перед x, то где-то здесь присутствует ошибка, т.к. все эти три ограничения невозможно выполнить вместе. В общем случае, если задача планирования с ограничениями предшествования содержит ориентированный цикл, то решения не существует. Для проверки на наличие таких ошибок необходимо уметь решать следующую задачу.

> ***Обнаружение ориентированных циклов.*** Содержит ли заданный орграф ориентированный цикл? Если да, нужно найти вершины такого цикла, от некоторой вершины до нее же.

Граф может содержать огромное количество циклов (см. упражнение 4.2.11), поэтому нас интересует лишь один цикл, а не все. Для планирования заданий и многих других приложений *необходимо* отсутствие ориентированных циклов, поэтому орграфы без них играют особую роль.

> **Определение.** *Ориентированный ациклический граф* (ОАГ) — это орграф без ориентированных циклов.

Значит, решение задачи обнаружения ориентированных циклов отвечает на следующий вопрос: *Является ли указанный орграф ациклическим*? Для такой задачи нетрудно разработать решение на основе поиска в глубину и того факта, что стек рекурсивных вызовов, используемый системой, представляет "текущий" ориентированный путь (как нить, позволяющая вернуться назад в правиле Тремо для лабиринтов). Если мы встретили ориентированное ребро v->w в вершину w, которая уже находится в стеке, то мы обнаружили цикл: ведь стек является признаком наличия ориентированного пути из w в v, и ребро v->w замыкает этот цикл. Более того, отсутствие таких *обратных ребер* означает, что граф ацикличен. Класс `DirectedCycle` в листинге 4.2.3 использует этот принцип для реализации API, приведенного на рис. 4.2.12.



```
public class  DirectedCycle

                    DirectedCycle(Digraph G)  конструктор для поиска циклов
           boolean  hasCycle()                содержит ли G ориентированный цикл?
Iterable<Integer>   cycle()                   вершины цикла (если он существует)
```

Рис. 4.2.12. *API для обнаружения циклов в орграфах*

Листинг 4.2.3. Нахождение ориентированного цикла

```
public class DirectedCycle
{
  private boolean[] marked;
  private int[] edgeTo;
  private Stack<Integer> cycle;       // вершины цикла (если он есть)
  private boolean[] onStack;          // вершины в стеке рекурсивных вызовов

  public DirectedCycle(Digraph G)
  {
    onStack = new boolean[G.V()];
    edgeTo  = new int[G.V()];
    marked  = new boolean[G.V()];
    for (int v = 0; v < G.V(); v++)
       if (!marked[v]) dfs(G, v);
  }

  private void dfs(Digraph G, int v)
  {
    onStack[v] = true;
    marked[v] = true;
    for (int w : G.adj(v))
      if (this.hasCycle()) return;
      else if (!marked[w])
      {  edgeTo[w] = v; dfs(G, w);  }
      else if (onStack[w])
      {
        cycle = new Stack<Integer>();
        for (int x = v; x != w; x = edgeTo[x])
          cycle.push(x);
        cycle.push(w);
        cycle.push(v);
      }
    onStack[v] = false;
  }
  public boolean hasCycle()
  {  return cycle != null;  }
  public Iterable<Integer> cycle()
  {  return cycle;  }
}
```

Этот класс добавляет к стандартному рекурсивному методу `dfs()` логический массив `onStack[]`, позволяющий отслеживать вершины, в которых рекурсивный вызов еще не завершен. Когда встречается ребро `v->w` в вершину `w`, которая находится в стеке, это значит, что обнаружен ориентированный цикл, и его можно восстановить, пройдя по ссылками из массива `onStack[]` (рис. 4.2.13).

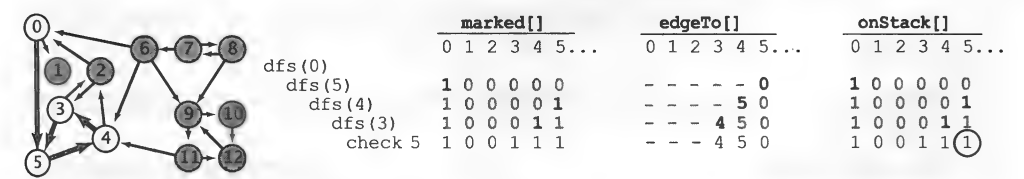


Рис. 4.2.13. Нахождение ориентированного цикла в орграфе

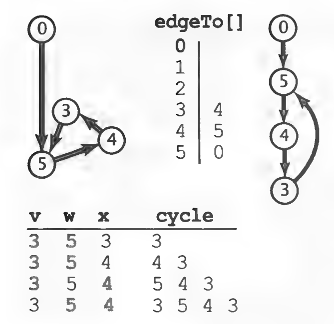


Рис. 4.2.14. Трассировка вычисления цикла

При выполнении вызова dfs(G, v) мы следуем по ориентированному пути от источника до вершины v. Для отслеживания этого пути в DirectedCycle используется индексированный вершинами массив onStack[] (рис. 4.2.14), который помечает вершины, находящиеся в стеке рекурсивных вызовов. Для этого элементу onStack[v] присваивается значение true при вызове dfs(G, v) и значение false при возврате. В классе DirectedCycle используется также массив edgeTo[], который позволяет вернуть цикл после его обнаружения — так же, как в классах DepthFirstPaths (листинг 4.1.5) и BreadthFirstPaths (листинг 4.1.6).

Упорядочивание поиском в глубину и топологическая сортировка

Планирование с ограничениями предшествования сводится к вычислению топологического порядка для вершин в ОАГ, как в API на рис. 4.2.15.

public class **Topological**	
Topological(Digraph G)	*конструктор для топологической сортировки*
boolean isDAG()	*является ли* **G** *ациклическим?*
Iterable<Integer> order()	*вершины в топологическом порядке*

Рис. 4.2.15. *API для топологического упорядочения*

Утверждение Д. Топологический порядок в орграфе возможен тогда и только тогда, когда он не содержит циклов.

Доказательство. Если в орграфе имеется ориентированный цикл, то в нем нет топологической упорядоченности. Иначе алгоритм, который мы сейчас рассмотрим, вычисляет топологическую упорядоченность для любого заданного ОАГ.

Оказывается, мы уже знакомы с алгоритмом топологической сортировки: одна дополнительная строка в нашем стандартном рекурсивном ПвГ делает все, что нужно! Чтобы убедиться в этом, мы начнем с класса DepthFirstOrder из листинга 4.2.4. Он основан на наблюдении, что поиск в глубину посещает каждую вершину только один раз. Если сохранять в структуре данных вершину, переданную в качестве аргумента в рекурсивный метод dfs(),то при переборе элементов этой структуры мы просмотрим все вершины графа, в порядке, который определяется природой структуры данных и сохранением данных до или после рекурсивных вызовов. В типичных приложениях обычно бывают нужны три вида упорядочения:

- *прямой порядок* (preorder) — вершины заносятся в очередь перед рекурсивными вызовами;
- *обратный порядок* (postorder) — вершины заносятся в очередь после рекурсивных вызовов;
- *реверсный порядок* (reverse postorder) — вершины заносятся в стек после рекурсивных вызовов.

Листинг 4.2.4. Упорядочение вершин в орграфе с помощью поиска в глубину

```
public class DepthFirstOrder
{
  private boolean[] marked;
  private Queue<Integer> pre;                // вершины в прямом порядке
  private Queue<Integer> post;               // вершины в обратном порядке
  private Stack<Integer> reversePost;        // вершины в реверсном порядке
  public DepthFirstOrder(Digraph G)
  {
    pre         = new Queue<Integer>();
    post        = new Queue<Integer>();
    reversePost = new Stack<Integer>();
    marked  = new boolean[G.V()];
    for (int v = 0; v < G.V(); v++)
       if (!marked[v]) dfs(G, v);
  }
  private void dfs(Digraph G, int v)
  {
    pre.enqueue(v);
    marked[v] = true;
    for (int w : G.adj(v))
      if (!marked[w])
        dfs(G, w);
    post.enqueue(v);
    reversePost.push(v);
  }
  public Iterable<Integer> pre()          {  return pre;  }
  public Iterable<Integer> post()         {  return post;  }
  public Iterable<Integer> reversePost()  {  return reversePost;  }
}
```

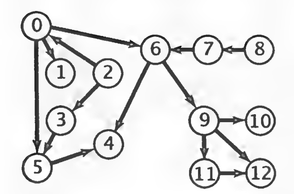
Этот класс позволяет клиентам перебирать вершины в различных порядках, определяемых поиском в глубину. Такая возможность очень полезна при разработке сложных алгоритмов обработки графов, т.к. рекурсивная природа поиска позволяет доказывать свойства вычислений — см., к примеру, утверждение Е.

Трассировка работы класса DepthFirstOrder для демонстрационного ОАГ приведена на рис. 4.2.16. Его нетрудно реализовать, и он поддерживает методы pre(), post() и reversePost(), которые часто бывают необходимы в более сложных алгоритмах обработки графов. Например, метод order() в классе Topological содержит просто вызов reversePost().

Утверждение Е. Реверсный порядок в ОАГ является топологическим порядком.

Доказательство. Рассмотрим ребро v->w. При вызове dfs(v) возможен один из трех случаев (рис. 4.2.17).

- Вызов dfs(w) уже был выполнен и вернул управление (w помечена).
- Вызов dfs(w) еще не был выполнен (w не помечена), поэтому v->w приводит к непосредственному или косвенному вызову dfs(w) (и возврату) до завершения работы dfs(v).
- Вызов dfs(w) уже был выполнен, но еще не завершил работу при выполнении dfs(v). Для доказательства важно, что этот случай невозможен в ОАГе, т.к. цепочка рекурсивных вызовов означает путь от w до v, и ребро v->w замыкает ориентированный цикл.



прямой порядок — это порядок вызовов `dfs()` → pre (*очередь*)

обратный порядок — это порядок обработки вершин → post (*очередь*)

reversePost (*стек*)

	pre	post	reversePost
dfs(0)	**0**		
dfs(5)	0 **5**		
dfs(4)	0 5 **4**		
4 done		4	**4**
5 done		4 **5**	**5** 4
dfs(1)	0 5 4 **1**		
1 done		4 5 1	**1** 5 4
dfs(6)	0 5 4 1 **6**		
dfs(9)	0 5 4 1 6 **9**		
dfs(11)	0 5 4 1 6 9 **11**		
dfs(12)	0 5 4 1 6 9 11 **12**		
12 done		4 5 1 **12**	**12** 1 5 4
11 done		4 5 1 12 **11**	**11** 12 1 5 4
dfs(10)	0 5 4 1 6 9 11 12 **10**		
10 done		4 5 1 12 11 **10**	**10** 11 12 1 5 4
check 12			
9 done		4 5 1 12 11 10 **9**	**9** 10 11 12 1 5 4
check 4			
6 done		4 5 1 12 11 10 9 **6**	**6** 9 10 11 12 1 5 4
0 done		4 5 1 12 11 10 9 6 **0**	**0** 6 9 10 11 12 1 5 4
check 1			
dfs(2)	0 5 4 1 6 9 11 12 10 **2**		
check 0			
dfs(3)	0 5 4 1 6 9 11 12 10 2 **3**		
check 5			
3 done		4 5 1 12 11 10 9 6 0 **3**	**3** 0 6 9 10 11 12 1 5 4
2 done			
check 3		4 5 1 12 11 10 9 6 0 3 **2**	**2** 3 0 6 9 10 11 12 1 5 4
check 4			
check 5			
check 6			
dfs(7)	0 5 4 1 6 9 11 12 10 2 3 **7**		
check 6			
7 done		4 5 1 12 11 10 9 6 0 3 2 **7**	**7** 2 3 0 6 9 10 11 12 1 5 4
dfs(8)	0 5 4 1 6 9 11 12 10 2 3 7 **8**		
check 7			
8 done		4 5 1 12 11 10 9 6 0 3 2 7 **8**	**8** 7 2 3 0 6 9 10 11 12 1 5 4
check 9			
check 10			
check 11			
check 12			

реверсный порядок (↑ reversePost)

Рис. 4.2.16. *Упорядочение вершин в орграфе (прямое, обратное и реверсное) с помощью поиска в глубину*

В двух возможных случаях вызов dfs(w) выполняется до dfs(v), и поэтому w появляется *перед* v в обратном порядке и *после* v в реверсном порядке. Значит, каждое ребро направлено от более ранней вершины к более поздней — что и требуется.

Класс Topological (алгоритм 4.5 в листинге 4.2.5) представляет собой реализацию, в которой выполняется топологическая сортировка ОАГ с помощью поиска в глубину. Трассировка его работы показана на рис. 4.2.17.

Листинг 4.2.5. Алгоритм 4.5. Топологическая сортировка

```
public class Topological
{
  private Iterable<Integer> order;      // топологический порядок

  public Topological(Digraph G)
  {
    DirectedCycle cyclefinder = new DirectedCycle(G);
    if (!cyclefinder.hasCycle())
    {
      DepthFirstOrder dfs = new DepthFirstOrder(G);
      order = dfs.reversePost();
    }
  }

  public Iterable<Integer> order()
  {  return order;  }

  public boolean isDAG()
  {  return order == null;  }

  public static void main(String[] args)
  {
    String filename = args[0];
    String separator = args[1];
    SymbolDigraph sg = new SymbolDigraph(filename, separator);

    Topological top = new Topological(sg.G());

    for (int v : top.order())
      StdOut.println(sg.name(v));
  }
}
```

Этот клиент классов DepthFirstOrder и DirectedCycle возвращает топологическое упорядочение ОАГ. Клиент тестирования решает задачу планирования с ограничениями предшествования для объекта SymbolDigraph. Метод экземпляра order() возвращает null, если указанный орграф содержит циклы, иначе итератор выдает вершины в топологическом порядке. Код SymbolDigraph опущен, т.к. он в точности совпадает с кодом SymbolGraph (листинг 4.1.10), с заменой всех идентификаторов Graph на Digraph.

```
% more jobs.txt
Algorithms/Theoretical CS/Databases/Scientific Computing
Introduction to CS/Advanced Programming/Algorithms
Advanced Programming/Scientific Computing
Scientific Computing/Computational Biology
```

```
Theoretical CS/Computational Biology/Artificial Intelligence
Linear Algebra/Theoretical CS
Calculus/Linear Algebra
Artificial Intelligence/Neural Networks/Robotics/Machine Learning
Machine Learning/Neural Networks

% java Topological jobs.txt "/"
Calculus
Linear Algebra
Introduction to CS
Advanced Programming
Algorithms
Theoretical CS
Artificial Intelligence
Robotics
Machine Learning
Neural Networks
Databases
Scientific Computing
Computational Biology
```

Утверждение Ж. Поиск в глубину позволяет выполнить топологическую сортировку ОАГ за время, пропорциональное $V+E$.

Доказательство. Непосредственно следует из кода. В нем используется поиск в глубину, который гарантирует отсутствие в графе ориентированных циклов, и еще один, который выполняет реверсное упорядочение. В обоих поисках просматриваются все ребра и все вершины, поэтому необходимое время выполнения пропорционально $V+E$.

Несмотря на простоту этого алгоритма, на него долго не обращали внимания, а вместо него был популярен более наглядный алгоритм с использованием очереди источников (см. упражнение 4.2.30).

На практике топологическое упорядочение и обнаружение циклов обычно работают совместно, причем обнаружение циклов играет роль отладочного средства. Например, в приложении планирования заданий ориентированный цикл в орграфе означает ошибку, которую необходимо исправить, как бы ни было сформулировано расписание выполнения. Поэтому приложение планирования заданий обычно выполняется в три этапа.

- Определяются задания и ограничения предшествования.
- Проверяется, существует ли решение: для этого в исходном орграфе находятся и удаляются циклы, пока не будут удалены все.
- Решается задача составления расписания с помощью топологической сортировки.

Аналогично можно проверять на появление циклов и все изменения в расписаниях (с помощью класса `DirectedCycle`), после чего вычислять новое расписание (с помощью класса `Topological`).

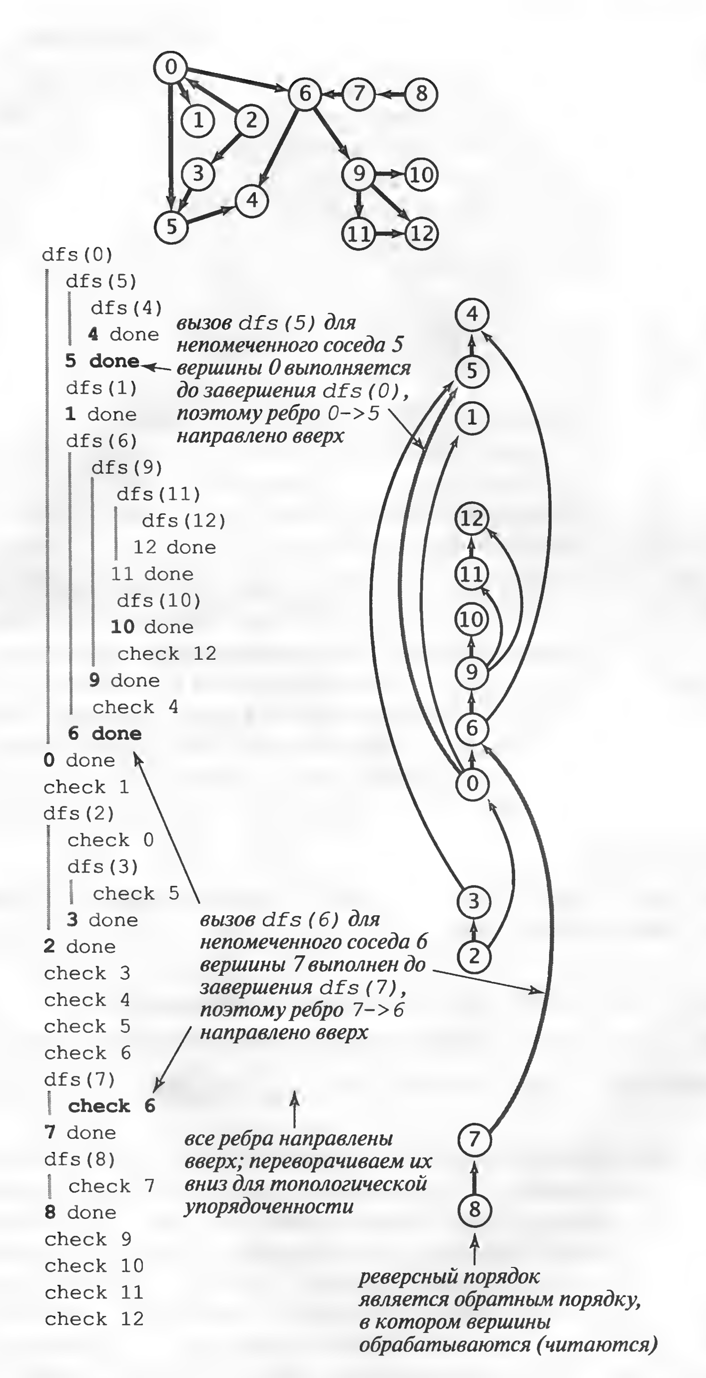


Рис. 4.2.17. Реверсный порядок в ОАГе — это топологическое упорядочение

Сильная связность в орграфах

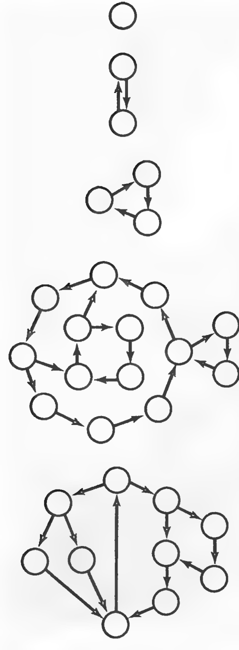


Рис. 4.2.18. Сильно связные орграфы

Мы старались всячески подчеркивать различие между достижимостью в орграфах и связностью в неориентированных графах. В неориентированном графе две вершины v и w связаны, если существует соединяющий их путь, и этот путь можно использовать как для перемещения из v в w, так и для перемещения из w в v. В орграфе вершина w достижима из v, если существует ориентированный путь из v в w, но ориентированный путь обратно из w в v может существовать, а может, и нет. В завершение нашего изучения орграфов мы рассмотрим естественный аналог связности в неориентированных графах.

Определение. Две вершины v и w являются *сильно связанными*, если они взаимно достижимы — т.е. существует ориентированный путь из v в w и ориентированный путь из w в v. Орграф называется *сильно связанным*, если все его вершины сильно связаны друг с другом.

Несколько примеров сильно связанных графов приведены на рис. 4.2.18. Как видно из этих примеров, в понимании сильной связности важную роль играют циклы. И действительно, если вспомнить, что ориентированный цикл общего вида представляет собой ориентированный цикл, в котором вершины могут повторяться, то легко видеть, что *две вершины сильно связаны тогда и только тогда, когда существует связывающий их ориентированный цикл общего вида.* (*Доказательство*: постройте пути из v в w и из w в v.)

Сильные компоненты

Как и связность в неориентированных графах, сильная связность в орграфах является отношением эквивалентности для множества вершин, т.к. она обладает следующими свойствами:

- *рефлексивность* — каждая вершина сильно связана с собой;
- *симметричность* — если v сильно связана с w, то и w сильно связана с v;
- *транзитивность* — если v сильно связана с w и w сильно связана с x, то v сильно связана с x.

В качестве отношения эквивалентности сильная связность разбивает вершины орграфа на классы эквивалентности. Классы эквивалентности — это максимальные подмножества вершин, сильно связанных между собой, и каждая вершина принадлежит в точности одному подмножеству. Такие подмножества мы будем называть *сильно связанными компонентами* или, более кратко, *сильными компонентами*. В нашем демонстрационном орграфе tinyDG.txt имеется пять сильных компонентов (рис. 4.2.19). Орграф с V вершинами может иметь от 1 до V сильных компонентов: сильно связанный орграф представляет собой 1 сильный компонент, а ОАГ содержит V сильных компонентов. Обратите внимание, что сильные компоненты определяются через вершины, а не ребра. Некоторые ребра соединяют две вершины в одном и том же сильном компоненте, а другие соединяют вершины из различных сильных компонентов. Последние не при-

надлежат ни одному ориентированному циклу. Выявление связных компонентов обычно важно для обработки неориентированных графов; аналогично и выявление сильных компонентов обычно важно при обработке орграфов.

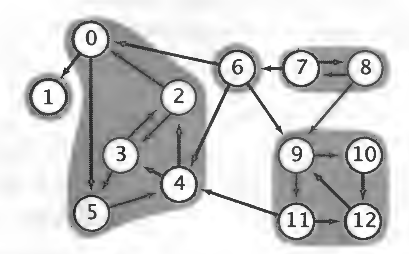


Рис. 4.2.19. Орграф и его сильные компоненты

Примеры применений

Сильная связность — полезная абстракция для понимания структуры орграфа, которая выявляет взаимосвязанные множества вершин (сильные компоненты). Например, сильные компоненты могут помочь авторам учебника сгруппировать темы, а разработчикам ПО — организовать программные модули (табл. 4.2.3).

Таблица 4.2.3. Типичные применения сильных компонентов

Область	Вершина	Ребро
Веб-сеть	Веб-страница	Гиперссылка
Учебник	Тема	Ссылка
Программное обеспечение	Модуль	Вызов
Пищевая сеть	Организм	Отношение хищник-жертва

На рис. 4.2.20 приведен пример из экологии: это орграф, моделирующий пищевую сеть, которая связывает живые организмы. Вершины соответствуют видам, а ребра из одной вершины в другую означают, что одни организмы питаются другими (*из* которых направлены ребра). Изучение таких орграфов (с тщательно выбранными множествами видов и тщательно отслеженными взаимосвязями) помогает экологам отвечать на базовые вопросы об экологических системах. На рис. 4.2.24 приведен орграф, моделирующий веб-контент, где вершины соответствуют страницам, а ребра — гиперссылкам из одной страницы на другую. Сильные компоненты в таких орграфах могут помочь сетевым инженерам разбить огром-

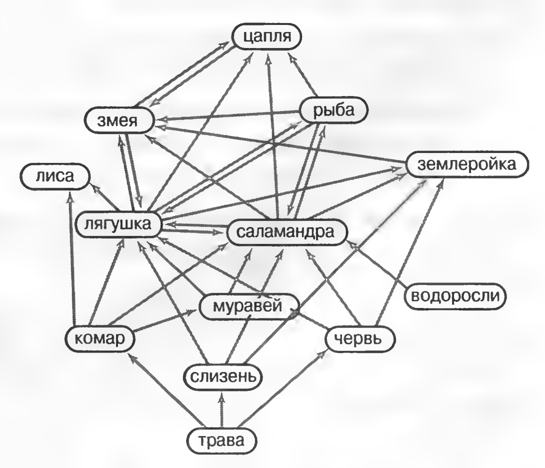


Рис. 4.2.20. Небольшое подмножество из орграфа пищевой сети

ное количество страниц в сети на подмножества более обозримых размеров, удобные для обработки. Дальнейшие свойства таких приложений и другие примеры содержатся в упражнениях и на сайте книги.

Так что нам понадобится API, аналогичный API CC для связных компонентов в неориентированных графах (рис. 4.1.27) — он приведен на рис. 4.2.21.

public class **SCC**		
	SCC(Digraph G)	*конструктор предобработки*
boolean	stronglyConnected(int v, int w)	*являются ли* v *и* w *сильно связанными?*
int	count()	*количество сильных компонентов*
int	id(int v)	*идентификатор компонента для* v *(от* 0 *до* count()-1*)*

Рис. 4.2.21. *API для сильных компонентов*

Для такого API нетрудно написать квадратичный алгоритм поиска сильных компонентов (см. упражнение 4.2.23), но (как обычно), квадратичное требование к времени и памяти не позволяет работать с очень большими графами, которые возникают в практических приложениях вроде только что описанных.

Алгоритм Косараю

В алгоритме CC (алгоритм 4.3 в листинге 4.1.8) мы видели, что вычисление связных компонентов в неориентированных графах сводится к простому применению поиска в глубину. А как можно эффективно находить сильные компоненты в орграфах? Такой поиск выполняет реализация KosarajuSCC (листинг 4.2.6), которая представляет собой алгоритм CC с добавлением нескольких дополнительных строк кода.

- Для заданного орграфа G с помощью экземпляра DepthFirstOrder находится реверсный порядок обращенного графа G^R.
- Для графа G выполняется обычный ПвГ, но непомеченные вершины выбираются в только что вычисленном порядке, а не в стандартном числовом.
- Все вершины, достижимые в конструкторе одним рекурсивным вызовом dfs(), принадлежат *одному сильному компоненту* (!), и их можно идентифицировать так же, как в CC.

Листинг 4.2.6. Алгоритм 4.6. Алгоритм Косараю для вычисления сильных компонентов

```
public class KosarajuSCC
{
   private boolean[] marked;          // вершины, до которых добрался поиск
   private int[] id;                  // идентификаторы компонентов
   private int count;                 // количество сильных компонентов
   public KosarajuSCC(Digraph G)
   {
      marked = new boolean[G.V()];
      id = new int[G.V()];
      DepthFirstOrder order = new DepthFirstOrder(G.reverse());
      for (int s : order.reversePost())
         if (!marked[s])
```

```
      {  dfs(G, s); count++;  }
   }
   private void dfs(Digraph G, int v)
   {
      marked[v] = true;
      id[v] = count;
      for (int w : G.adj(v))
         if (!marked[w])
            dfs(G, w);
   }
   public boolean stronglyConnected(int v, int w)
   {  return id[v] == id[w];  }
   public int id(int v)
   {  return id[v];  }
   public int count()
   {  return count;  }
}
```

Эта реализация отличается от CC (алгоритм 4.3) только выделенными фрагментами (и в реализации метода main(), где используется код из листинга 4.1.7, но с заменой Graph на Digraph и CC на KosarajuSCC). Для поиска сильных компонентов алгоритм выполняет поиск в глубину в обращенном орграфе, чтобы получить упорядочение вершин (реверсный порядок в этом поиске), которое затем используется при поиске в глубину в исходном орграфе.

```
% java KosarajuSCC tinyDG.txt
5 components
1
0 5 4 3 2
11 12 9 10
6
8 7
```

Алгоритм Косараю — яркий пример метода, который легко написать, но трудно понять. Несмотря на его загадочность, вы можете внимательно проследить ход доказательства следующего утверждения, используя рис. 4.2.22, и убедиться в его корректности.

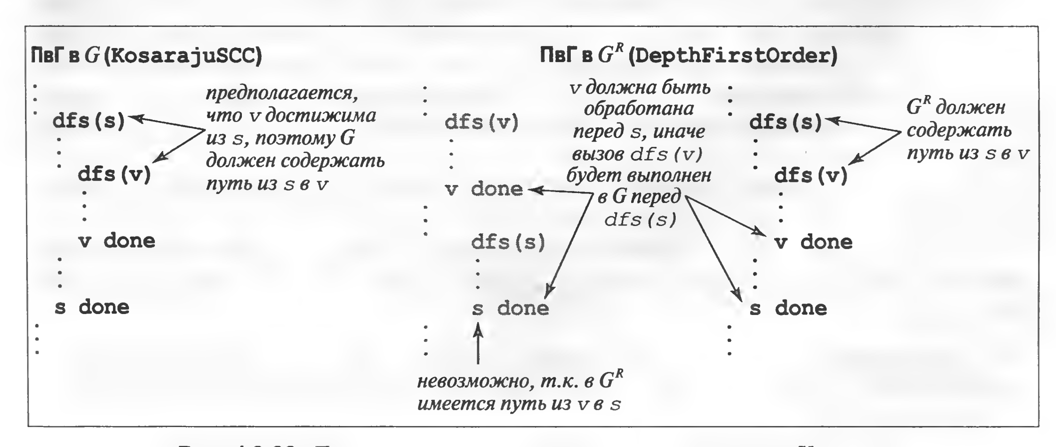


Рис. 4.2.22. Доказательство корректности алгоритма Косараю

Утверждение З. В ПвГ на орграфе G, где помеченные вершины выбираются в реверсном порядке, заданном с помощью ПвГ на обращенном орграфе G^R (алгоритм Косараю), вершины, до которых доходит каждый вызов рекурсивного метода из конструктора, принадлежат одному сильному компоненту.

Доказательство. Вначале мы докажем от противного, что *каждая вершина* v, *сильно связанная с* s, *достижима в вызове конструктора* dfs(G, s). Предположим, что вершина v, сильно связанная с s, не достижима в вызове dfs(G, s). В силу наличия пути из s в v вершина v уже должна быть помечена. Но в силу наличия пути из v в s вершина s должна быть помечена при вызове dfs(G, v), и конструктор не выполнил бы вызов dfs(G, s). Полученное противоречие доказывает исходное положение.

Теперь мы докажем, что *каждая вершина* v, *достижимая в вызове* dfs(G, s) *из конструктора, сильно связана с* s. Пусть v — вершина, до которой дошел вызов dfs(G, s). Тогда существует путь в G из s в v, и остается только доказать, что в G существует путь из v в s. Это утверждение эквивалентно утверждению, что в G^R существует путь из s в v, и остается только доказать, что в G^R действительно существует путь из s в v.

Основной момент — доказательство, что из конструкции реверсного порядка следует, что вызов dfs(G, v) должен быть выполнен до dfs(G, s) во время ПвГ в G^R. Это оставляет на рассмотрение лишь два случая для dfs(G, v): он может быть выполнен:

- *до* вызова dfs(G, s) (и завершен перед этим вызовом);
- *после* вызова dfs(G, s) (и завершен до завершения dfs(G, s)).

Первый вариант невозможен, т.к. в G^R существует путь из v в s, а из второго варианта следует, что в G^R существует путь из s в v, что и завершает доказательство.

Трассировка алгоритма Косараю для файла tinyDG.txt приведена на рис. 4.2.23. Справа от каждой трассировки ПвГ приведен чертеж орграфа, на котором вершины представлены в порядке их обработки. Просмотр чертежа реверсного орграфа (слева) дает реверсное упорядочение — порядок, в котором непомеченные вершины проверяются в ПвГ в исходном орграфе. Как видно из диаграммы, второй ПвГ выполняет вызов dfs(1) (который помечает вершину 1), потом вызов dfs(0) (который помечает вершины 5, 4, 3 и 2), затем проверяет вершины 2, 4, 5 и 3, далее выполняет вызов dfs(11) (который помечает вершины 11, 12, 9 и 10), затем проверяет вершины 9, 12 и 10, потом вызывает dfs(6) (который помечает вершину 6) и, наконец, вызывает dfs(7) (который помечает 7 и 8).

Более крупный пример, в котором используется очень небольшое подмножество модели орграфа для веб-сети, приведен на рис. 4.2.24.

Алгоритм Косараю решает следующий аналог задачи связности для неориентированных графов, которая была сначала сформулирована в главе 1, а потом мы еще вернулись к ней в разделе 4.1.

Сильная связность. Для заданного орграфа нужны ответы на запросы вида *Являются ли две указанные вершины сильно связанными?* и *Сколько сильных компонентов содержит орграф?*

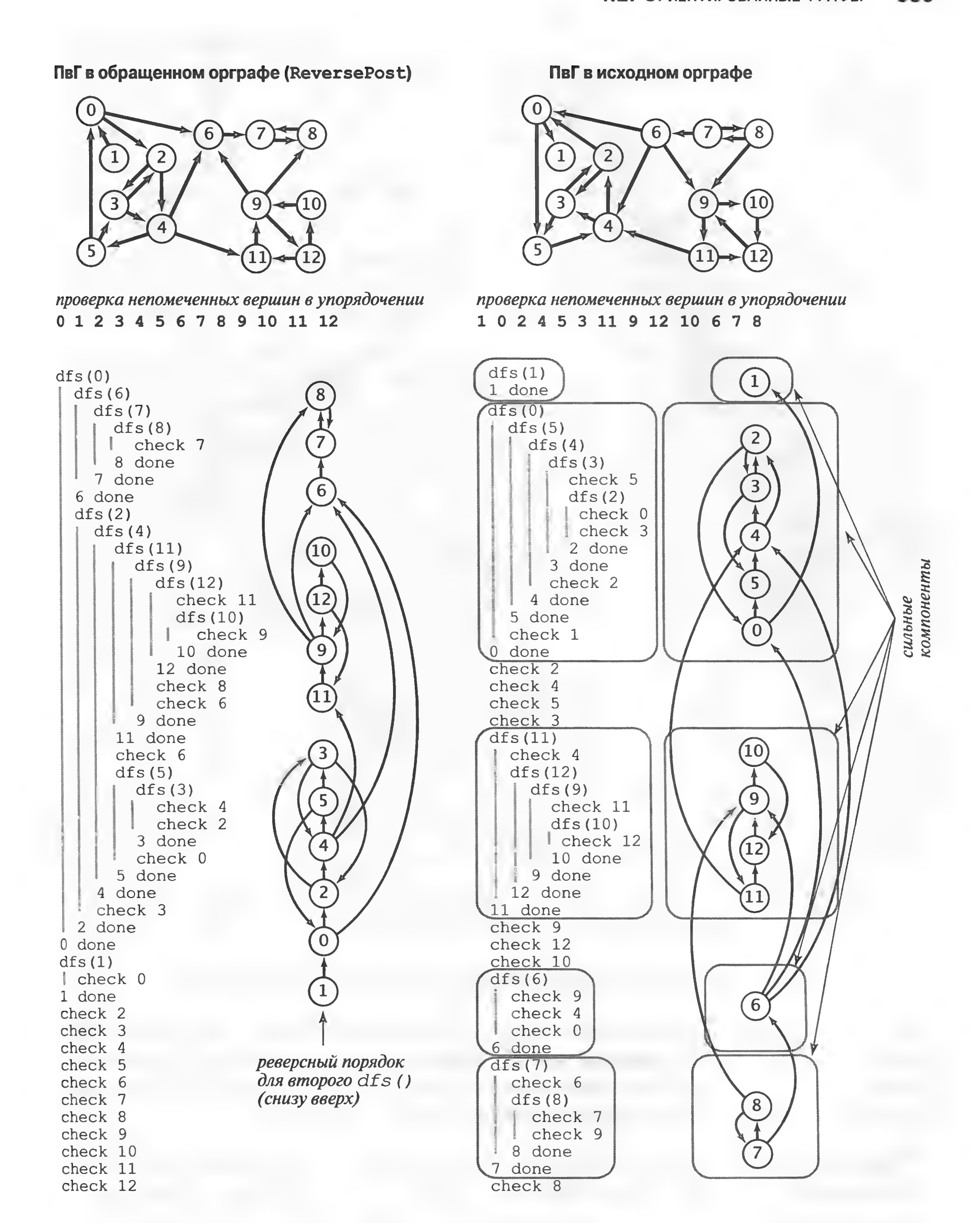


Рис. 4.2.23. Алгоритм Косараю для нахождения сильных компонентов в орграфах

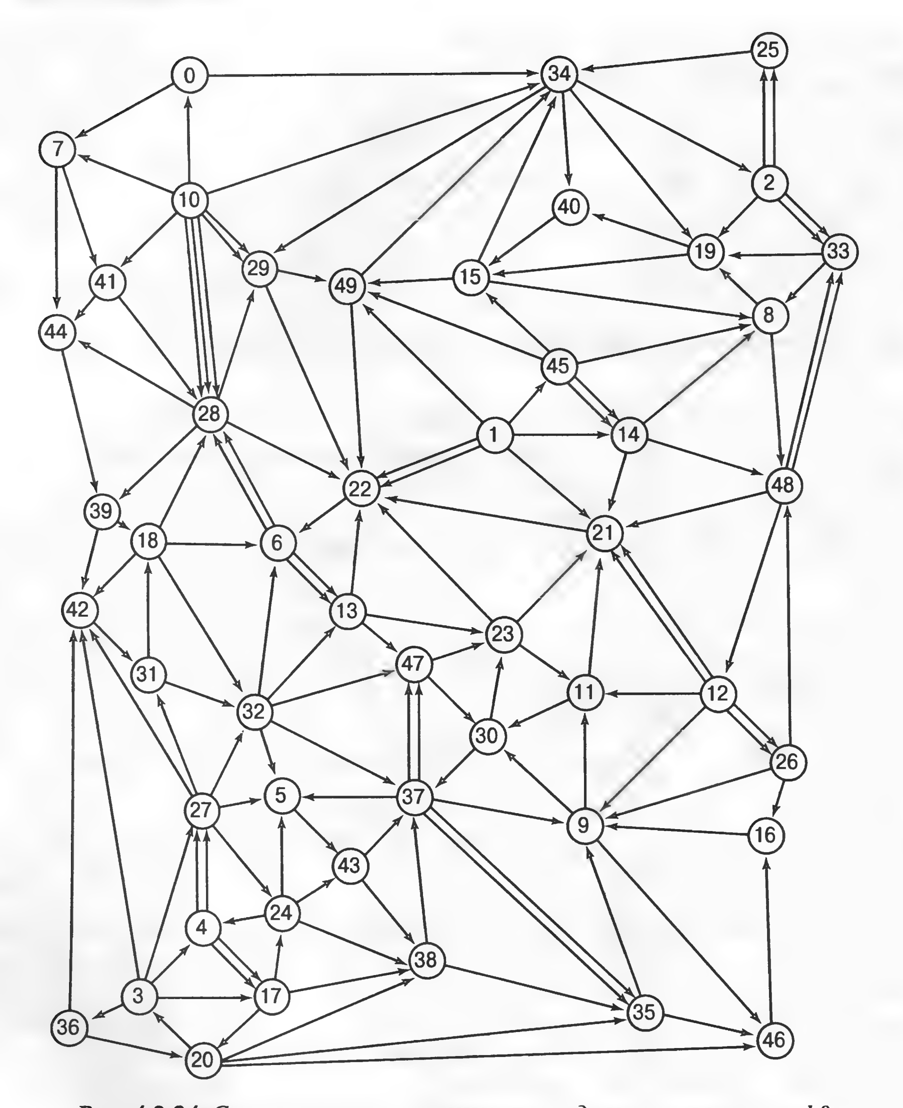


Рис. 4.2.24. *Сколько сильных компонентов содержит этот орграф?*

Можно ли решить эту задачу для орграфов так же эффективно, как соответствующую задачу связности в неориентированных графах — одно время ответ на данный вопрос был неясен (решение было найдено Р.Е. Тарьяном (Tarjan) в конце 1970-х годов). Удивительно, что полученное решение оказалось настолько простым.

Утверждение И. Для ответов на запросы о сильной связности в орграфах алгоритм Косараю использует время предобработки и объем памяти, пропорциональные $V+E$.

Доказательство. Алгоритм вычисляет обращение орграфа и выполняет два поиска в глубину. Каждый из этих шагов выполняется за время, пропорциональное $V+E$. Обращенная копия орграфа занимает объем памяти, пропорциональный $V+E$.

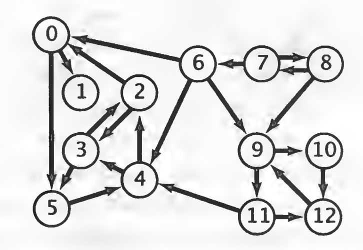
Еще раз о достижимости

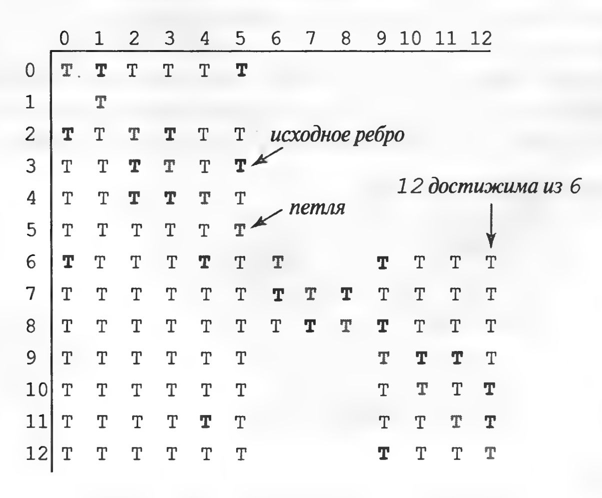
Реализация `CC` для неориентированных графов позволяет установить, связаны ли вершины `v` и `w`. Если они связаны, то существует путь из `v` и `w` и (тот же самый) путь из `w` в `v`. Реализация `KosarajuCC` позволяет установить сильную связность вершин `v` и `w`, т.е. наличие пути из `v` и `w` и (другого) пути из `w` в `v`. А вот в отношении пар вершин, которые не являются сильно связными, такого сказать нельзя. Может быть путь из `v` в `w` или путь из `w` в `v`, но не оба.

> ***Достижимость всех пар.*** Для заданного орграфа нужны ответы на запросы вида *Существует ли ориентированный путь из указанной вершины* `v` *в другую указанную вершину* `w`*?*

Для неориентированных графов соответствующая задача эквивалентна задаче связности, но для орграфов она весьма отличается от задачи сильной связности. Нашей реализации `CC` необходимо линейное время на предобработку, чтобы затем отвечать на подобные запросы за константное время. Возможна ли такая производительность для орграфов? На этот с виду несложный вопрос эксперты не могли ответить несколько десятков лет. Чтобы лучше понять, в чем трудность, рассмотрим диаграмму на рис. 4.2.25, которая иллюстрирует следующую фундаментальную концепцию.

Определение. *Транзитивным замыканием* орграфа *G* называется другой орграф с тем же множеством вершин, но ребро из вершины `v` в вершину `w` существует тогда и только тогда, когда `w` достижима из `v` в графе *G*.





	0	1	2	3	4	5	6	7	8	9	10	11	12
0	T	T	T	T	T	T							
1		T											
2	T	T	T	T	T	T							
3	T	T	T	T	T	T							
4	T	T	T	T	T	T							
5	T	T	T	T	T	T							
6	T	T	T	T	T	T	T			T	T	T	T
7	T	T	T	T	T	T	T	T	T	T	T	T	T
8	T	T	T	T	T	T	T	T	T	T	T	T	T
9	T	T	T	T	T	T				T	T	T	T
10	T	T	T	T	T	T				T	T	T	T
11	T	T	T	T	T	T				T	T	T	T
12	T	T	T	T	T	T				T	T	T	T

Рис. 4.2.25. Транзитивное замыкание

По соглашению каждая вершина достижима из себя, поэтому транзитивное замыкание содержит V петель. В нашем демонстрационном графе всего 13 ориентированных ребер, но его транзитивное замыкание содержит 102 из возможных 169 ориентированных ребер. Как правило, в транзитивном замыкании орграфа гораздо больше ребер, чем в самом орграфе, и совсем не редко разреженный граф имеет насыщенное транзитивное замыкание. Например, транзитивное замыкание ориентированного цикла с V вершинами и V ориентированными ребрами является полным орграфом с V^2 ориентированными ребрами. В силу того, что транзитивные замыкания обычно являются насыщенными, мы будем представлять их с помощью матриц логических значений, где элемент в строке `v` и столбце `w` равен `true` в том и только том случае, когда вершина `w` достижима из `v`. Вместо явного вычисления транзитивного замыкания можно реализовать API достижимости (рис. 4.2.26) с помощью поиска в глубину.

public class **TransitiveClosure**		
	`TransitiveClosure(Digraph G)`	*конструктор предобработки*
`boolean`	`reachable(int v, int w)`	*достижима ли* `w` *из* `v`*?*

***Рис. 4.2.26.** API для достижимости всех пар*

Код в листинге 4.2.7 содержит примитивную реализацию, в которой используется класс `DirectedDFS` (алгоритм 4.4). Это решение идеально для небольших или насыщенных графов, но не годится для огромных орграфов, которые могут встретиться на практике. Ведь *конструктор использует объем памяти, пропорциональный* V^2, *и время, пропорциональное* $V(V + E)$: каждый из V объектов `DirectedDFS` занимает память, пропорциональную V (содержит массив `marked[]` размером V и проверяет E ребер для расстановки пометок). По сути, класс `TransitiveClosure` вычисляет и сохраняет транзитивное замыкание графа G, чтобы отвечать за константное время на запросы: строка `v` из матрицы транзитивного замыкания эквивалентна массиву `marked[]` для `v`-го элемента массива `DirectedDFS[]` из объекта `TransitiveClosure`. Можно ли обеспечить константное время запроса при значительно меньшем времени предобработки и значительно меньших затратах памяти? Общее решение, отвечающее на запросы за константное время с существенно меньшими, чем квадратичные, затратами памяти — до сих пор не решенная исследовательская задача. Эта задача может иметь важные практические следствия: например, пока она не решена, мы не можем надеяться получить практическое решение для задачи достижимости всех пар в гигантском графе наподобие всемирной компьютерной сети.

Листинг 4.2.7. Достижимость всех пар

```
public class TransitiveClosure
{
   private DirectedDFS[] all;
   TransitiveClosure(Digraph G)
   {
      all = new DirectedDFS[G.V()];
      for (int v = 0; v < G.V(); v++)
         all[v] = new DirectedDFS(G, v);
   }
   boolean reachable(int v, int w)
   {  return all[v].marked(w);  }
```

Резюме

В данном разделе мы узнали об ориентированных ребрах и орграфах, о взаимосвязи между обработкой орграфов и соответствующих задачах для неориентированных графах и о других темах.

- Терминология для описания орграфов.
- Представление и принципы работы почти не отличаются от неориентированных графов, но некоторые задачи для орграфов решаются сложнее.
- Циклы, ОАГ, топологическая сортировка и планирование с ограничениями предшествования.
- Достижимость, пути и сильная связность в орграфах.

В табл. 4.2.4 содержится сводка по реализациям изученных нами алгоритмов на орграфах (все они, кроме одного, основаны на поиске в глубину). Все рассмотренные задачи имеют простые формулировки, но решения могут быть как легкими адаптациями соответствующих алгоритмов для неориентированных графов, так и хитроумными и неожиданными. Эти алгоритмы сами являются отправными точками для нескольких еще более сложных алгоритмов, которые будут рассмотрены в разделе 4.4, когда мы будем знакомиться с *взвешенными* орграфами.

Таблица 4.2.4. Задачи обработки орграфов, рассмотренные в данном разделе

Задача	Решение	Где
Достижимость из одного и нескольких источников	`DirectedDFS`	Листинг 4.2.2
Ориентированные пути из одного источника	`DepthFirstDirectedPaths`	Раздел "Поиск путей в орграфах"
Кратчайшие ориентированные пути из одного источника	`BreadthFirstDirectedPaths`	Раздел "Поиск путей в орграфах"
Обнаружение ориентированных циклов	`DirectedCycle`	Листинг 4.2.3
Упорядочение вершин поиском в глубину	`DepthFirstOrder`	Листинг 4.2.4
Планирование с ограничениями предшествования	`Topological`	Листинг 4.2.5
Топологическая сортировка	`Topological`	Листинг 4.2.5
Сильная связность	`KosarajuSCC`	Листинг 4.2.6
Достижимость для всех пар	`TransitiveClosure`	Листинг 4.2.7

Вопросы и ответы

Вопрос. **Является ли петля циклом?**

Ответ. Да, но для достижимости вершины из самой себя петли не нужны.

Упражнения

4.2.1. Каково максимальное количество ребер в орграфе с V вершинами и без параллельных ребер? Каково минимальное количество ребер в орграфе с V вершинами, ни одна из которых не является изолированной?

4.2.2. Нарисуйте, в стиле рис. 4.1.10, списки смежности, построенные конструктором из входного потока в классе `Digraph` для файла `tinyDGex2.txt`, приведенного на рис. 4.2.27.

4.2.3. Напишите конструктор копирования для класса `Digraph`, который принимает в качестве аргумента орграф `G` и создает и инициализирует новую копию орграфа. Любые изменения, выполненные после этого клиентом в `G`, не должны влиять на созданную копию.

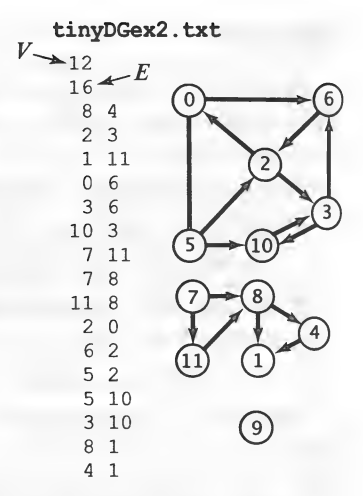


Рис. 4.2.27. Орграф для упражнения 4.2.2

4.2.4. Добавьте в класс `Digraph` метод `hasEdge()`, который принимает два целочисленных аргумента `v` и `w` и возвращает `true`, если в графе есть ребро `v->w`, иначе `false`.

4.2.5. Добавьте в класс `Digraph` запрет на существование параллельных ребер и петель.

4.2.6. Разработайте клиент тестирования для класса `Digraph`.

4.2.7. *Полустепень захода* (indegree) вершины в орграфе — это количество ориентированных ребер, которые направлены в данную вершину. *Полустепень исхода* вершины (outdegree) — это количество ориентированных ребер, которые направлены из данной вершины. Ни одна вершина не достижима из вершины с полустепенью исхода 0 — такая вершина называется *стоком*. Вершина с полустепенью захода 0 не достижима из любой другой вершины — такая вершина называется *источником*. Орграф, в котором разрешены петли, и полустепень исхода каждой вершины равна 1, называется *отображением* — функцией из множества целых чисел от 0 до $V-1$ в это же множество. Напишите программу `Degrees.java`, которая реализует API, приведенный на рис. 4.2.28.

public class **Degrees**		
	`Degrees(Digraph G)`	*конструктор*
`int`	`indegree(int v)`	*полустепень захода* `v`
`int`	`outdegree(int v)`	*полустепень исхода* `v`
`Iterable<Integer>`	`sources()`	*источники*
`Iterable<Integer>`	`sinks()`	*стоки*
`boolean`	`isMap()`	*является ли* `G` *отображением?*

Рис. 4.2.28. API для упражнения 4.2.7

4.2.8. Начертите все неизоморфные ОАГ с двумя, тремя, четырьмя и пятью вершинами (см. упражнение 4.1.28).

4.2.9. Напишите метод, который проверяет, является ли заданная перестановка вершин ОАГ топологическим порядком этого ОАГ.

4.2.10. Пусть задан ОАГ; существует ли топологический порядок, который невозможно получить с помощью алгоритма на основе ПвГ, независимо от порядка выбор вершин, смежных с каждой вершиной? Обоснуйте свой ответ.

4.2.11. Опишите семейство разреженных орграфов, в котором количество ориентированных циклов растет экспоненциально в зависимости от количества вершин.

4.2.12. Сколько ребер содержит транзитивное замыкание орграфа, который представляет собой простой ориентированный путь с V вершинами и $V-1$ ребрами?

4.2.13. Приведите транзитивное замыкание для орграфа с десятью вершинами и ребрами

```
3->7  1->4  7->8  0->5  5->2  3->8  2->9  0->6  4->9  2->6  6->4
```

4.2.14. Докажите, что сильные компоненты в G^R совпадают с сильными компонентами из G.

4.2.15. Что представляют собой сильные компоненты ориентированного ациклического графа?

4.2.16. Что произойдет, если выполнить алгоритм Косараю для ОАГ?

4.2.17. Верно ли, что реверсный порядок в обращении графа совпадает с обратным порядком в исходном графе?

4.2.18. Подсчитайте объем памяти, необходимой для работы объекта `Digraph` с V вершинами и E ребрами в условиях модели стоимости памяти из раздела 1.4.

Творческие задачи

4.2.19. *Топологическая сортировка и ПвШ.* Объясните, почему следующий алгоритм не обязательно приведет к появлению топологического порядка: выполняется ПвШ, и вершины маркируются в порядке увеличения расстояния до их соответствующего источника.

4.2.20. *Ориентированный эйлеров цикл.* Эйлеров цикл — это ориентированный цикл, содержащий каждое ребро графа в точности один раз. Напишите клиент `Euler`, который обнаруживает эйлеров цикл или сообщает, что такой цикл не существует. *Совет*: докажите, что орграф G содержит эйлеров цикл тогда и только тогда, когда G является связным, и полустепень захода каждой вершины равна ее полустепени исхода.

4.2.21. *Ближайшие общие предки в ОАГ.* Пусть заданы ОАГ и две вершины `v` и `w`. Нужно найти *ближайшего общего предка* (БОП) этих вершин. БОП вершин `v` и `w` — это общий предок вершин `v` и `w`, такой, что у него нет потомков, которые также были бы предками `v` и `w`. Вычисление БОП полезно при множественном наследовании в языках программирования, анализе генеалогических данных (определение степени близкородственного скрещивания в графе родословной) и в других приложениях. *Совет*: определите высоту вершины `v` в ОАГ как длину самого длинного пути из корня в `v`. Среди вершин, которые являются предками и `v`, и `w`, вершина с наибольшей высотой и будет ближайшим общим предком `v` и `w`.

4.2.22. *Кратчайший наследственный путь.* Пусть заданы ОАГ и две вершины v и w. Нужно найти *кратчайший наследственный путь* между v и w. Наследственный путь между v и w — это общий предок x вместе с кратчайшим путем из v в x и кратчайшим путем из w в x. Кратчайший наследственный путь — это наследственный путь с минимальной общей длиной. *Разминка*: найдите ОАГ, в котором кратчайший наследственный путь приводит к общему предку, который не является ближайшим общим предком. *Совет*: выполните ПвШ два раза, один раз для v и еще раз для w.

4.2.23. *Сильный компонент.* Опишите алгоритм с линейным временем выполнения для вычисления сильно связанного компонента, содержащего заданную вершину v. Используя этот алгоритм, опишите простой квадратичный алгоритм для вычисления сильных компонентов в орграфе.

4.2.24. *Гамильтонов путь в ОАГ.* Пусть задан ОАГ. Предложите линейный алгоритм для определения, существует ли ориентированный путь, который проходит через каждую вершину в точности один раз.

Ответ: выполните топологическую сортировку и проверьте, существует ли ребро между каждой последовательной парой вершин в топологическом порядке.

4.2.25. *Уникальное топологическое упорядочение.* Разработайте алгоритм для определения, имеет ли орграф уникальное топологическое упорядочение. *Совет*: орграф имеет уникальное топологическое упорядочение в том и только в том случае, когда существует ориентированное ребро между каждой парой последовательных вершин в топологическом порядке (т.е. в орграфе имеется гамильтонов путь). Если орграф имеет несколько топологических упорядочений, то следующее топологическое упорядочение можно получить, обменяв пару последовательных вершин.

4.2.26. *2-выполнимость.* Пусть задана логическая формула в конъюнктивной нормальной форме с M термами и N литералами, такая, что каждый терм содержит точно два литерала, и нужно найти удовлетворяющее присваивание (если оно возможно). *Совет*: постройте *орграф импликации* с $2N$ вершинами (по одной для литерала и его отрицания). Для каждого терма $x \vee y$ добавьте ребра от $\neg y$ к x и от $\neg x$ к y. Для истинности терма $x \vee y$ необходимо, чтобы (1) если y ложно, то x истинно, или (2) если x ложно, то y истинно. *Требование*: формула выполнима тогда и только тогда, когда ни одна переменная x не находится в том же сильном компоненте, что и ее отрицание $\neg x$. А топологическая сортировка *ядерного ОАГ* (когда каждый сильный компонент стягивается в единую вершину) дает выполнимое присваивание.

4.2.27. *Подсчет орграфов.* Покажите, что количество различных орграфов с V вершинами без параллельных ребер равно 2^{V^2}. (Сколько может быть орграфов, содержащих V вершин и E ребер?) Затем вычислите верхнюю границу процента орграфов с 20 вершинами, которые может просмотреть "вселенский компьютер". При этом предположите, что каждый электрон во вселенной может просмотреть орграф за одну наносекунду, во вселенной не более 10^{80} электронов, а возраст вселенной не превышает 10^{20} лет.

4.2.28. *Подсчет ОАГов.* Приведите формулу для количества ОАГ с V вершинами и E ребрами.

4.2.29. *Арифметические выражения.* Напишите класс, который вычисляет значения ОАГ, представляющих арифметические выражения. Для хранения значений, соответствующих каждой вершине, используйте индексированный вершинами массив. Считайте, что значения, соответствующие листьями, заданы. Опишите семейство таких арифметических выражений, что размер дерева выражения экспоненциально больше, чем размер соответствующего ОАГ (т.е. время работы программы для ОАГ пропорционально логарифму времени выполнения для дерева).

4.2.30. *Топологическая сортировка с помощью очереди.* Напишите реализацию топологической сортировки, в которой используется индексированный вершинами массив для отслеживания полустепени захода каждой вершины. Выполните инициализацию массива и очереди источников единым проходом по всем ребрам, как в упражнении 4.2.7. Затем выполняйте следующие операции, пока очередь источников не станет пуста.

- Удалите источник из очереди и пометьте его.
- Уменьшите на 1 значения элементов в массиве полустепеней захода, соответствующих вершинам назначения каждого из ребер удаленной вершины.
- Если после уменьшения какого-либо элемента он стал равен 0, вставьте соответствующую вершину в очередь источников.

4.2.31. *Евклидовы орграфы.* На основе решения упражнения 4.1.37 создайте API `EuclideanDigraph` для графов, вершины которого представляют собой точки на плоскости, чтобы можно было работать с графическим представлением.

Эксперименты

4.2.32. *Случайные орграфы.* Напишите программу `ErdosRenyiDigraph`, которая принимает из командной строки целочисленные значения V и E и строит орграф, генерируя E случайных пар целых чисел от 0 до $V-1$. *Внимание*: такой генератор может создавать петли и параллельные ребра.

4.2.33. *Случайные простые орграфы.* Напишите программу `RandomDigraph`, которая принимает из командной строки целочисленные значения V и E и строит с одинаковой вероятностью один из возможных *простых* орграфов с V вершинами и E ребрами.

4.2.34. *Случайные разреженные орграфы.* Напишите на основе решения упражнения 4.1.41 программу `RandomSparseDigraph`, которая генерирует случайные разреженные орграфы для подобранного множества значений V и E, которые можно использовать для выполнения осмысленных эмпирических тестов.

4.2.35. *Случайные евклидовы орграфы.* Напишите на основе решения упражнения 4.1.42 клиент `RandomEuclideanDigraph` программы `EuclideanDigraph`, который присваивает ребрам случайные направления.

4.2.36. *Случайные орграфы на решетке.* Напишите на основе решения упражнения 4.1.43 клиент `RandomGridDigraph` программы `EuclideanDiGraph`, который присваивает ребрам случайные направления.

4.2.37. *Реальные орграфы.* Найдите в Интернете орграф большого размера — возможно, граф транзакций в какой-то онлайновой системе или орграф, определяемый ссылками на веб-страницы. Напишите программу `RandomRealDigraph`, которая строит граф, выбирая случайным образом V вершин и E ориентированных ребер из подграфа, индуцированного этими вершинами.

4.2.38. *Реальные ОАГ.* Найдите в Интернете ОАГ большого размера — возможно, определенный зависимостями из определений классов в большой программной системе или ссылками на каталоги в большой файловой системе. Напишите программу `RandomRealDAG`, которая строит граф, выбирая случайным образом V вершин и E ориентированных ребер из подграфа, индуцированного этими вершинами.

Тестирование всех алгоритмов со всеми возможными параметрами на всех моделях графов выполнить нереально. Для каждой из приведенных ниже задач напишите клиент, решающий эту задачу, а затем выберите один из описанных выше генераторов, чтобы выполнять эксперименты для данной модели графов. Планируйте эксперименты обдуманно, возможно, на основе результатов предыдущих экспериментов. Напишите краткий анализ полученных результатов и выводы из этих результатов.

4.2.39. *Достижимость.* Эмпирически определите среднее количество вершин, достижимых из произвольно выбранной вершины, для различных моделей орграфов.

4.2.40. *Длины путей в ПвГ.* Эмпирически определите вероятность того, что `DepthFirstDirectedPaths` найдет путь между двумя случайно выбранными вершинами, и среднюю длину соответствующих путей, для различных моделей орграфов.

4.2.41. *Длины путей в ПвШ.* Эмпирически определите вероятность того, что `BreadthFirstDirectedPaths` найдет путь между двумя случайно выбранными вершинами, и среднюю длину соответствующих путей, для различных моделей орграфов.

4.2.42. *Сильные компоненты.* Эмпирически найдите распределение количества сильных компонентов в случайных орграфах различных видов. Сгенерируйте большое количество орграфов и нарисуйте гистограмму.

4.3. Минимальные остовные деревья

Граф с взвешенными ребрами — это граф, в котором с каждым ребром связан *вес* или *стоимость*. Такие графы естественным образом моделируют многие ситуации. На карте авиасообщения, где ребра представляют авиарейсы, эти веса могут представлять расстояния или стоимости билетов. В электрической схеме, где ребра представляют проводники, веса могут представлять длину проводника, его стоимость или время прохождения сигнала по нему. В таких ситуациях естественно возникает вопрос минимизации стоимости. В данном разделе мы рассмотрим *неориентированные* графы с взвешенными ребрами и алгоритмы решения одной задачи.

Минимальное остовное дерево. Для заданного неориентированного графа с взвешенными ребрами нужно найти МОД.

Определение. Напомним, что *остовное дерево* графа — это связный подграф без циклов, который содержит все вершины графа. *Минимальное остовное дерево* (МОД) графа с взвешенными ребрами — это остовное дерево, вес которого (сумма весов его ребер) не больше веса любого другого остовного дерева (рис. 4.3.1).

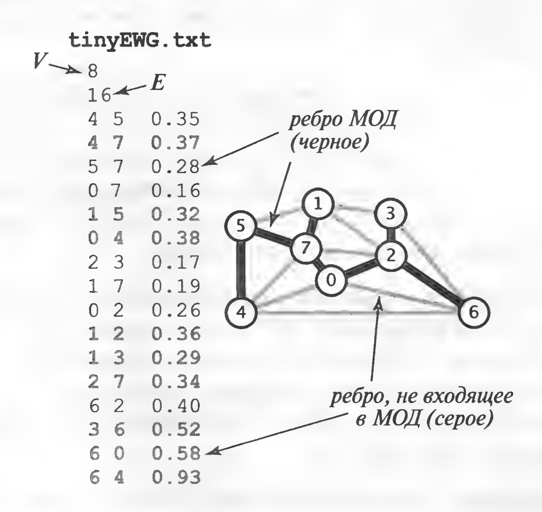


Рис. 4.3.1. Граф с взвешенными ребрами и его МОД

В данном разделе мы изучим два классических алгоритма для вычисления МОД: *алгоритм Прима* и *алгоритм Крускала*. Эти алгоритмы легко понять и нетрудно реализовать; они входят в число самых старых и наиболее известных алгоритмов из этой книги, однако они хорошо согласуются и с современными структурами данных. В силу многочисленных важных применений МОД алгоритмы для решения этой задачи изучались, по меньшей мере, с 1920-х годов — сначала в контексте электросетей, а затем в контексте телефонных сетей. В настоящее время на алгоритмах нахождения МОД основано построение многих видов сетей (коммуникационные, электрические, гидравлические, компьютерные, дорожные, железнодорожные, воздушные и многие другие), а также изучение биологических, химических и физических сетей, которые встречаются в природе (табл. 4.3.1).

18 Зак. 3239

Таблица 4.3.1. Типичные применения МОД

Область	Вершина	Ребро
Микросхема	Компонент	Проводник
Авиасообщение	Аэропорт	Авиарейс
Электросети	Электростанция	Линии электропередачи
Анализ изображений	Образ	Отношение близости

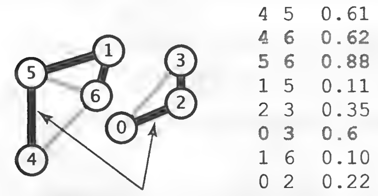
Соглашения

При вычислении минимальных остовных деревьев могут возникнуть различные аномальные ситуации (рис. 4.3.2), которые обычно легко устраняются. Чтобы упростить изложение, мы будем придерживаться перечисленных ниже соглашений.

- *Граф является связным.* Из определения остовного дерева следует, что для его существования необходима связность графа. Задачу можно сформулировать и подругому, если вспомнить базовые свойства деревьев из раздела 4.1: нужно найти множество $V-1$ ребер с минимальным весом, которое соединяет все вершины графа. Если граф не связен, то можно вычислить МОД для каждого из его связных компонентов, которые в таком случае все вместе называются *минимальным остовным лесом* (см. упражнение 4.3.22).
- *Веса ребер не обязательно означают расстояние.* Иногда геометрическая наглядность помогает лучше понять алгоритмы, поэтому мы будем использовать примеры, где вершины представляются точками на плоскости, а веса ребер — расстояниями между ними, как на рис. 4.3.1. Однако важно помнить, что веса могут соответствовать времени, стоимости или какой-то другой переменной и совсем не обязаны быть пропорциональными расстояниям.
- *Веса ребер могут быть нулевыми или отрицательными.* Если веса всех ребер положительны, достаточно определить МОД как подграф с минимальным общим весом, который соединяет все вершины, поскольку такой подграф должен иметь вид остовного дерева. Однако условие остовного дерева уже включено в определение, поэтому такое определение применимо для графов, ребра которых могут иметь нулевые или отрицательные веса.
- *Все веса ребер различны.* При наличии ребер с одинаковыми весами минимальное остовное дерево может и не быть уникальным (см. упражнение 4.3.2). Возможность наличия нескольких МОД усложняет доказательства корректности некоторых алгоритмов, поэтому мы подавим эту возможность в зародыше. Вообще-то это ограничение не такое уж и жесткое, т.к. наши алгоритмы работают без всяких изменений и при наличии одинаковых весов.

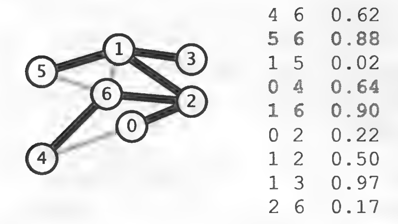
В общем, мы будем считать, что требуется найти МОД для связного графа с ребрами, которым назначены произвольные (но различные) веса

МОД не существует, если граф не связен

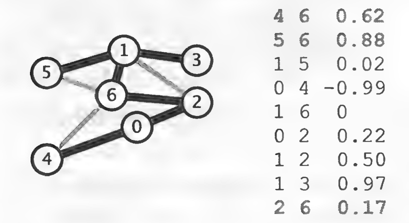


можно независимо вычислить МОД компонентов

Веса не обязаны быть пропорциональны расстояниям



Веса могут быть нулевыми или отрицательными



МОД может быть не уникальным при наличии одинаковых весов

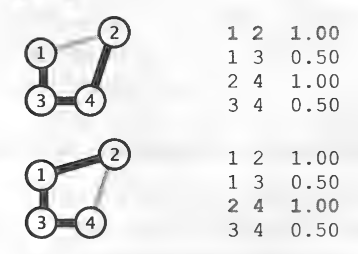


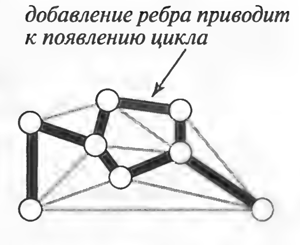
Рис. 4.3.2. Различные аномалии МОД

Базовые принципы

Сначала мы вспомним два определяющих свойства дерева из раздела 4.1 (рис. 4.3.3).

- Добавление в дерево ребра, соединяющего две вершины, приводит к появлению уникального цикла.
- Удаление ребра из дерева разбивает его на два отдельных поддерева.

Эти свойства лежат в основе доказательства фундаментального свойства МОД, на котором базируются алгоритмы, рассматриваемые в данном разделе.



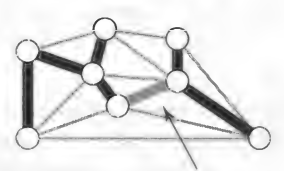


Рис. 4.3.3. Основные свойства дерева

Свойство сечения

Это свойство — *свойство сечения* — помогает выявить ребра, которые должны присутствовать в МОД заданного графа с взвешенными ребрами. Для этого выполняется разбиение вершин дерева на два множества и проверка ребер, которые пересекают это разбиение.

Определение. *Сечение* графа — это разбиение его вершин на два непустых раздельных множества. *Перекрестное ребро* сечения — это ребро, соединяющее вершину из одного множества с вершиной из другого множества.

Обычно мы будем задавать сечение, указывая некоторое множество вершин и неявно предполагая, что сечение состоит из указанного множества и его дополнения. Поэтому перекрестное множества — это ребро из вершины, которая принадлежит множеству, в вершину, которая не принадлежит этому множеству. На рисунках вершины одной стороны сечения мы будем изображать серыми кружочками, а вершины другой стороны — белыми кружочками.

Утверждение К (свойство сечения). Для любого сечения в графе с взвешенными ребрами перекрестное ребро минимального веса принадлежит МОД этого графа.

Доказательство. Пусть e — перекрестное ребро минимального веса, и пусть T — МОД. Проведем доказательство от противного: допустим, что T не содержит e. Рассмотрим граф, образованный добавлением e к T. В этом графе имеется цикл, содержащий e, и этот цикл должен содержать как минимум еще одно перекрестное ребро — скажем, f — вес которого превышает вес e (т.к. e минимально, а все веса ребер различны). Тогда можно получить остовное дерево строго меньшего веса, удалив из него f и добавив e — полученное противоречие доказывает минимальность T.

Если (по соглашению) веса всех ребер различны, то у каждого связного графа имеется уникальное МОД (см. упражнение 4.3.3), и свойство сечения утверждает, что кратчайшее ребро из любого сечения должно входить в МОД.

На рис. 4.3.4 свойство сечения изображено графически. Учтите, что минимальное ребро не обязательно должно быть *единственным* ребром МОД, соединяющим два множества — типичные сечения обычно содержат несколько ребер МОД, которые соединяют вершину из одного множества с вершиной из другого множества (рис. 4.3.5).

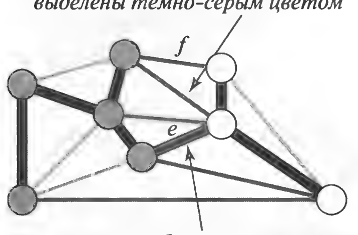


Рис. 4.3.4. Свойство сечения

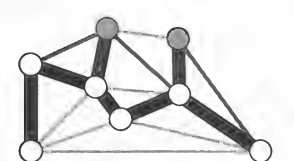


Рис. 4.3.5. Сечение с двумя ребрами МОД

"Жадный" алгоритм

На свойстве сечения основаны алгоритмы нахождения МОД. Вообще-то они являются частными случаями общей парадигмы, которая называется *"жадным" алгоритмом*: с помощью свойства сечения находится ребро, входящее в МОД, и так повторяется, пока не будут найдены все ребра МОД. Наши алгоритмы различаются способами использования сечений и поиска перекрестного ребра минимального веса, но это разновидности следующего принципа.

Утверждение Л ("жадный" алгоритм МОД). Следующий метод закрашивает черным цветом все ребра МОД любого связного графа с взвешенными ребрами и V вершинами: вначале закрашиваем все ребра серым цветом, потом находим сечение без черных ребер, закрашиваем его ребро с минимальным весом в черный цвет, и продолжаем так, пока не будут закрашены черным $V-1$ ребер.

Доказательство. Для простоты предположим, что веса всех ребер различны, хотя утверждение верно и если это не так (см. упражнение 4.3.5). По свойству сечения любое ребро, закрашенное черным цветом, принадлежит МОД. Если черным цветом закрашено менее $V-1$ ребер, существует сечение без черных ребер (ведь предполагается, что граф связный). Когда будут закрашены $V-1$ ребер, черные ребра образуют остовное дерево.

На рис. 4.3.6 приведена типичная трассировка "жадного" алгоритма. На каждой диаграмме показано сечение и ребро с минимальным весом в этом сечении (жирное красное), которое добавлено алгоритмом в МОД.

Тип данных для графа с взвешенными ребрами

Как можно представлять графы с взвешенными ребрами в программах? Пожалуй, проще всего слегка расширить базовое представление графа из раздела 4.1: в матрицах смежности можно хранить не логические значения, а веса ребер; в списках смежности можно определить узел, чтобы он содержал не только вершину, но и поле веса. (Как обычно, мы будем работать с разреженными графами и оставим представление матрицей смежности на проработку в упражнениях.)

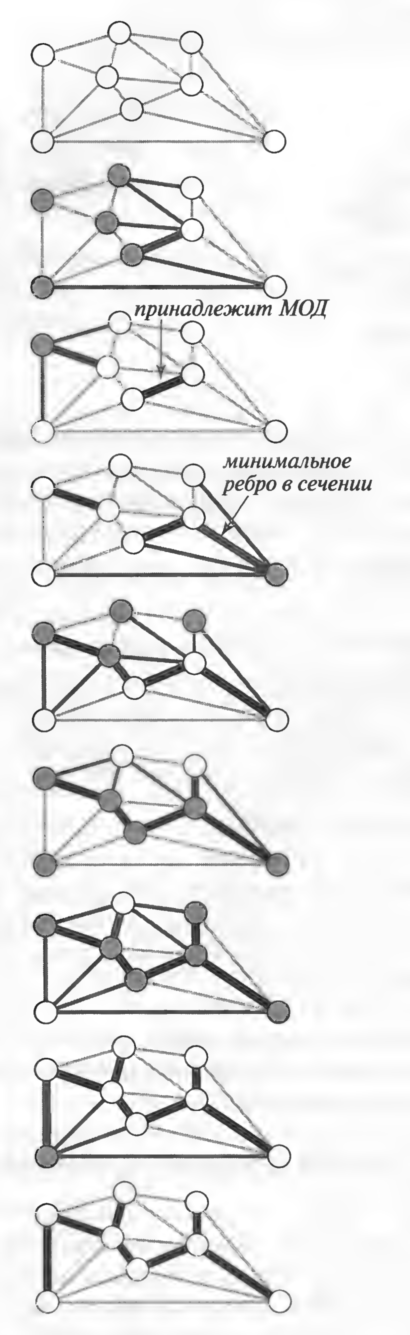


Рис. 4.3.6. "Жадный" алгоритм построения МОД

Этот классический способ вполне пригоден, но мы воспользуемся другим способом, который не намного сложнее, но позволит работать нашим программам в более общих контекстах. Для этого способа потребуется несколько более общий API, который позволит работать с объектами ребер, как показано на рис. 4.3.7.

public class **Edge** implements Comparable<Edge>

	Edge(int v, int w, double weight)	*конструктор инициализации*
double	weight()	*вес данного ребра*
int	either()	*одна из вершин данного ребра*
int	other(int v)	*другая вершина*
int	compareTo(Edge that)	*сравнение данного ребра с* that
String	toString()	*строковое представление*

Рис. 4.3.7. API для взвешенного ребра

Методы either() и other() для доступа к вершинам ребра поначалу выглядят странновато, но их необходимость станет понятна при рассмотрении кода клиентов. Реализация типа Edge приведена в листинге 4.3.1. На ее основе будет разработан API EdgeWeightedGraph, в котором естественным образом задействованы объекты Edge (рис. 4.3.8).

Листинг 4.3.1. Тип данных для взвешенного ребра

```
public class  Edge implements Comparable<Edge>
{
  private final int v;                         // одна вершина
  private final int w;                         // другая вершина
  private final double weight;                 // вес ребра
  public Edge(int v, int w, double weight)
  {
     this.v = v;
     this.w = w;
     this.weight = weight;
  }
  public double weight()
  {  return weight;  }
  public int either()
  {  return v;  }
  public int other(int vertex)
  {
     if      (vertex == v) return w;
     else if (vertex == w) return v;
     else throw new RuntimeException("Недопустимое ребро");
  }
  public int compareTo(Edge that)
  {
     if     (this.weight() < that.weight())  return -1;
     else if (this.weight() > that.weight()) return +1;
     else                                    return  0;
  }
  public String toString()
  {  return String.format("%d-%d %.2f", v, w, weight);  }
}
```

Этот тип данных содержит методы either() и other(), которые позволяют клиентам использовать вызов вроде other(v), чтобы найти другую вершину, если известна v. Если неизвестна ни одна вершина, то для доступа к обеим вершинам ребра e в клиентах будет использоваться код int v = e.either(), w = e.other(v);.

public class **EdgeWeightedGraph**		
	EdgeWeightedGraph(int V)	*создание пустого графа с V вершинами*
	EdgeWeightedGraph(In in)	*чтение графа из входного потока* in
int	V()	*количество вершин*
int	E()	*количество ребер*
void	addEdge(Edge e)	*добавление в граф ребра* e
Iterable<Edge>	adj(int v)	*ребра, инцидентные* v
Iterable<Edge>	adj(int v)	*все ребра графа*
String	toString()	*строковое представление*

Рис. 4.3.8. *API для графа с взвешенными ребрами*

Этот API очень похож на API для типа Graph (рис. 4.1.8). Но имеются и два ощутимых различия: он основан на типе Edge, и в нем имеется дополнительный метод edges() (см. листинг 4.3.2), который позволяет клиентам перебирать все ребра графа (игнорируя петли). Остальная часть реализации EdgeWeightedGraph в листинге 4.3.3 весьма похожа на реализацию неориентированного графа без весов из раздела 4.1 — только в списках смежности вместо целых чисел используются объекты Edge.

Листинг 4.3.2. Сбор всех ребер в графе с взвешенными ребрами

```
public Iterable<Edge> edges()
{
   Bag<Edge> b = new Bag<Edge>();
   for (int v = 0; v < V; v++)
      for (Edge e : adj[v])
         if (e.other(v) > v) b.add(e);
   return b;
}
```

Листинг 4.3.3. Тип данных для графа с взвешенными ребрами

```
public class EdgeWeightedGraph
{
   private final int V;                  // количество вершин
   private int E;                        // количество ребер
   private Bag<Edge>[] adj;              // списки смежности

   public EdgeWeightedGraph(int V)
   {
      this.V = V;
      this.E = 0;
      adj = (Bag<Edge>[]) new Bag[V];
```

```
    for (int v = 0; v < V; v++)
       adj[v] = new Bag<Edge>();
  }
  public EdgeWeightedGraph(In in)
  // См. упражнение 4.3.9.
  public int V() {  return V;  }
  public int E() {  return E;  }
  public void addEdge(Edge e)
  {
    int v = e.either(), w = e.other(v);
    adj[v].add(e);
    adj[w].add(e);
    E++;
  }
  public Iterable<Edge> adj(int v)
  {  return adj[v];  }
  public Iterable<Edge> edges()
 // См. листинг 4.3.2.
}
```

В этой реализации используются индексированные вершинами списки ребер. Как и в типе Graph (см. листинг 4.1.1), каждое ребро представлено дважды: если оно соединяет вершины v и w, то оно присутствует и в списке вершины v, и в списке вершины w. Метод edges() помещает все ребра в контейнер Bag (см. листинг 4.3.2). Реализация toString() оставлена в качестве упражнения.

На рис. 4.3.9 показано представление графа с взвешенными ребрами, построенное конструктором EdgeWeightedGraph для демонстрационного файла tinyEWG.txt, и содержимое каждого объекта Bag оформлено в виде связного списка, в соответствии со стандартной реализацией из раздела 1.3. Чтобы не загромождать рисунок, каждый объект Edge представлен в виде пары значений: целочисленного и вещественного. Реальная структура данных — это связный список ссылок на объекты, содержащие эти значения. В частности, несмотря на наличие двух *ссылок* на каждый объект Edge (по одной в списке для каждой вершины), каждому ребру графа соответствует только один объект Edge. На рисунке ребра содержатся в списках в порядке, обратном порядку их обработки — из-за похожего на стек поведения стандартной реализации связного списка. Как и в классе Graph, использование типа Bag подчеркивает, что клиентский код не делает никаких предположений о порядке объектов в списках.

Сравнение весов ребер

В API указано, что класс Edge должен реализовать интерфейс Comparable и содержать реализацию compareTo(). В графе с упорядоченными ребрами естественно упорядочивать ребра по их весам. Из этого следует примитивная реализация метода compareTo().

Параллельные ребра

Как и в реализациях графов с неориентированными ребрами, здесь разрешены параллельные ребра. Либо можно разработать более сложную реализацию EdgeWeightedGraph, где они запрещены — возможно, храня ребро с минимальным весом из всего множества параллельных ребер.

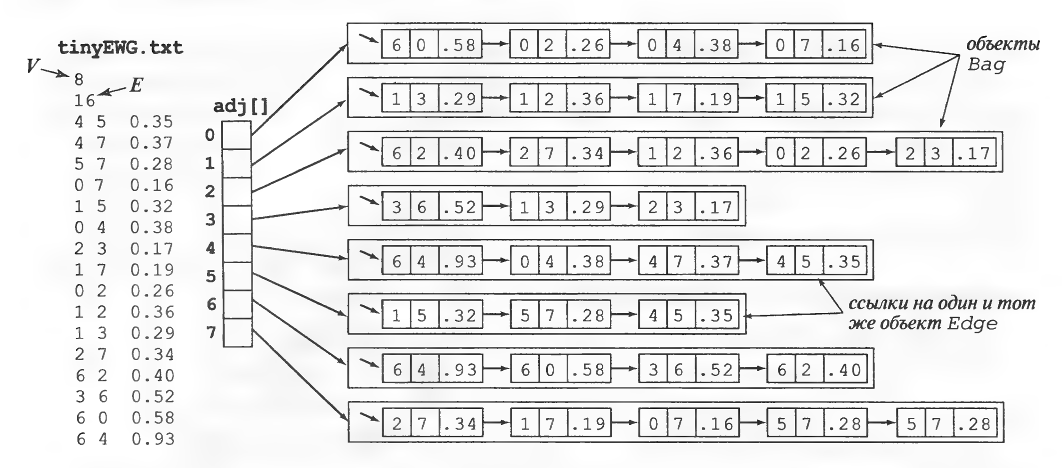


Рис. 4.3.9. Представление графа с взвешенными ребрами

Петли

Наличие петель допускается. Однако наша реализация `edges()` в классе `EdgeWeightedGraph` не содержит петель, хотя они могут встретиться во входных данных или в структуре данных. Это допущение не влияет на поведение алгоритмов поиска МОД, т.к. ни одно МОД не содержит петель. При работе с теми приложениями, где петли важны, вам необходимо соответственно изменить код.

Наше решение использовать явные объекты `Edge` приводит к ясному и компактному клиентскому коду. Это достигается небольшой ценой: каждый узел списков смежности содержит *ссылку* на объект `Edge`, с избыточной информацией (все узлы из списка смежности вершины содержат и эту вершину). Кроме того, увеличиваются затраты из-за использования объектов. У нас используется лишь одна копия каждого объекта `Edge`, но на него указывают две ссылки. Можно сделать и по-другому: использовать два узла списков, соответствующих каждому ребру (как в классе `Graph`) — с другой вершиной и весом ребра в каждом из этих узлов. Но за это тоже надо платить: два узла с двумя копиями веса для каждого ребра.

API МОД и клиент тестирования

Как обычно в обработке графов, мы определим API, где конструктор принимает в качестве аргумента граф с взвешенными ребрами и поддерживает клиентские методы запросов, которые возвращают МОД и его вес. А как представить само МОД? Минимальное остовное дерево графа — это его подграф, который является деревом, поэтому можно выбрать один из нескольких вариантов. Основные из них:

- список ребер;
- граф с взвешенными ребрами;
- индексированный вершинами массив с родительскими ссылками.

Чтобы предоставить клиентам и нашим реализациям максимальную свободу в выборе этих альтернатив, мы будем придерживаться API, приведенного на рис. 4.3.10.

public class **MST**		
	MST(EdgeWeightedGraph G)	*конструктор*
Iterable<Edge>	edges()	*все ребра МОД*
double	weight()	*вес МОД*

Рис. 4.3.10. *API для реализаций МОД*

Клиент тестирования

Как обычно, мы создадим демонстрационные графы и разработаем клиент тестирования для проверки работоспособности наших реализаций. Пример клиента приведен в листинге 4.3.4. Он считывает ребра из входного потока, строит граф с взвешенными ребрами, вычисляет МОД этого графа, выводит ребра МОД и выводит общий вес МОД.

Листинг 4.3.4. Клиент тестирования МОД

```
public static void main(String[] args)
{
  In in = new In(args[0]);
  EdgeWeightedGraph G;
  G = new EdgeWeightedGraph(in);
  MST mst = new MST(G);
  for (Edge e : mst.edges())
    StdOut.println(e);
  StdOut.println(mst.weight());
}
```

```
% more tinyEWG.txt
8 16
4 5 .35
4 7 .37
5 7 .28
0 7 .16
1 5 .32
0 4 .38
2 3 .17
1 7 .19
0 2 .26
1 2 .36
1 3 .29
2 7 .34
6 2 .40
3 6 .52
6 0 .58
6 4 .93

% java MST tinyEWG.txt
0-7 0.16
1-7 0.19
0-2 0.26
2-3 0.17
5-7 0.28
4-5 0.35
6-2 0.40
1.81
```

Рис. 4.3.11. *Пример графа с 8 вершинами и 16 ребрами*

Тестовые данные

На сайте книги содержится файл `tinyEWG.txt` (рис. 4.3.11), определяющий небольшой демонстрационный граф с рис. 4.3.1, который мы будем использовать для подробных трассировок алгоритмов нахождения МОД. На сайте книги есть и файл `mediumEWG.txt`, определяющий взвешенный граф с 250 вершинами — он показан на рис. 4.3.12. Это пример *евклидова графа*: его вершины являются точками на плоскости, а ребра — отрезки, соединяющие эти ребра, с весами, равными расстояниям между вершинами. Такие графы помогают глубже понять поведение алгоритмов МОД; кроме того, они моделируют многие уже упомянутые практические задачи, наподобие дорожных карт или электронных схем. На сайте книги доступен еще один, большой, файл `largeEWG.txt`, который определяет евклидов граф с 1 миллионом вершин. Наша цель — найти способ поиска МОД в таком графе за приемлемое время.

```
% more mediumEWG.txt
250 1273
244 246 0.11712
239 240 0.10616
238 245 0.06142
235 238 0.07048
233 240 0.07634
232 248 0.10223
231 248 0.10699
229 249 0.10098
228 241 0.01473
226 231 0.07638
... [ и еще 1263 ребер ]

% java MST mediumEWG.txt
  0 225 0.02383
 49 225 0.03314
 44  49 0.02107
 44 204 0.01774
 49  97 0.03121
202 204 0.04207
176 202 0.04299
176 191 0.02089
 68 176 0.04396
 58  68 0.04795
... [ и еще 239 ребер ]
10.46351
```

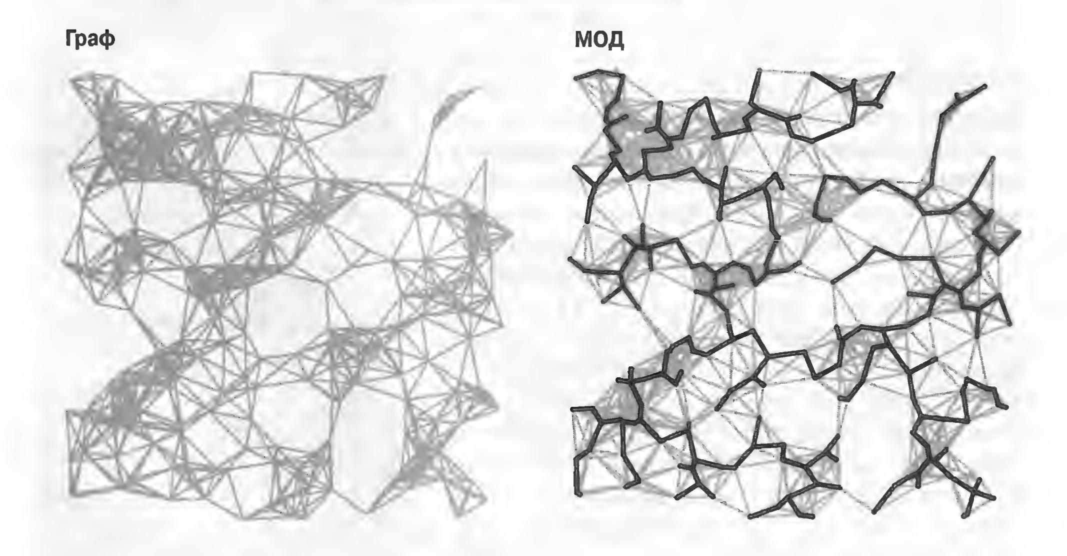


Рис. 4.3.12. Граф с 250 вершинами (и 1273 ребрами) и его МОД

Алгоритм Прима

Наш первый метод называется *алгоритмом Прима* (Prim). На каждом шаге своего выполнения он присоединяет новое ребро к единственному растущему дереву. Процесс начинается с любой вершины, которая считается первоначальным деревом, а потом к ней добавляются $V-1$ ребер (рис. 4.3.13). Следующим для добавления всегда выбирается (закрашенное на рисунке черным) ребро с минимальным весом, которое соединяет вершину дерева с вершиной, которая еще не принадлежит дереву (перекрестное ребро для сечения, определяемого вершинами дерева).

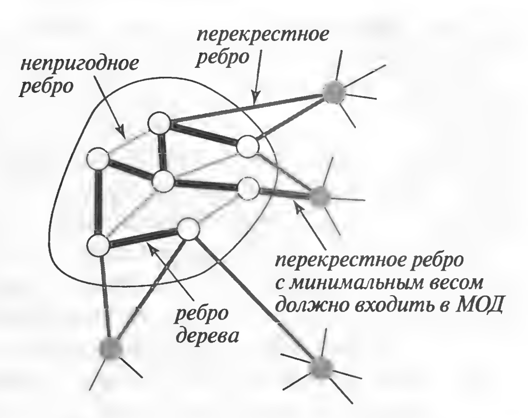


Рис. 4.3.13. *Алгоритм Прима для нахождения МОД*

Утверждение М. Алгоритм Прима вычисляет МОД любого связного графа с взвешенными ребрами.

Доказательство. Непосредственно следует из утверждения Л. Растущее дерево определяет сечение, в котором нет черных ребер; алгоритм выбирает перекрестное ребро минимального веса, поэтому он последовательно закрашивает ребра черным цветом в соответствии с "жадным" алгоритмом.

Вышеприведенное описание алгоритма Прима не дает ответа на следующий вопрос: как (эффективно) найти перекрестное ребро минимального веса. Было предложено несколько способов, и мы рассмотрим некоторые из них, но после разработки полного решения на основе крайне простого подхода.

Структуры данных

Мы реализуем алгоритм Прима с помощью нескольких простых и знакомых структур данных. В частности, мы будем представлять вершины дерева, ребра дерева и перекрестные ребра следующим образом.

- *Вершины дерева*: индексированный вершинами логический массив `marked[]`, где `marked[v]` равно `true`, если `v` находится в дереве.
- *Ребра дерева*: мы будем использовать две структуры данных — очередь `mst` для сбора ребер МОД или индексированный вершинами массив `edgeTo[]` объектов `Edge`, где `edgeTo[v]` содержит ребро, соединяющее `v` с деревом.
- *Перекрестные ребра*: очередь с приоритетами `MinPQ<Edge>`, которая сравнивает ребра по их весам (см. листинг 4.3.1).

Эти структуры данных позволяют непосредственно ответить на базовый вопрос: "Какое из перекрестных ребер имеет минимальный вес?"

Работа с множеством перекрестных ребер

При каждом добавлении ребра в дерево туда добавляется и вершина. Для работы с множеством перекрестных ребер необходимо добавить в очередь с приоритетами все ребра из данной вершины до всех вершин, которые не включены в дерево (для опре-

деления таких ребер предназначен массив marked[]). Но это еще не все: любое ребро, соединяющее только что добавленную вершину с вершиной дерева, которая находится в очереди с приоритетами, становится *непригодным* (уже не может быть перекрестным ребром, т.к. оно соединяет две вершины дерева). *"Энергичная"* реализация алгоритма Прима сразу удаляет такие ребра из очереди с приоритетами. Но мы сначала рассмотрим *"ленивую"* реализацию, где такие ребра остаются в очереди с приоритетами, а при их извлечении выполняется проверка на пригодность.

На рис. 4.3.14 приведена трассировка работы с нашим маленьким демонстрационным графом tinyEWG.txt. Каждая диаграмма содержит рисунок графа и очередь с приоритетами сразу после посещения вершины (т.е. после добавления в дерево и обработки ее списка смежности). Тут же приводится и содержимое очереди с приоритетами, где новые ребра помечены звездочками. Алгоритм строит МОД следующим образом.

- Добавляет вершину 0 в МОД, а все ребра из ее списка смежности — в очередь с приоритетами.
- Добавляет вершину 7 и ребро 0-7 в МОД, а все ребра из списка смежности 7 — в очередь с приоритетами.
- Добавляет вершину 1 и ребро 1-7 в МОД, а все ребра из списка смежности 1 — в очередь с приоритетами.
- Добавляет вершину 2 и ребро 0-2 в МОД, а ребра 2-3 и 6-2 — в очередь с приоритетами. Ребра 2-7 и 1-2 помечаются как непригодные.
- Добавляет вершину 3 и ребро 2-3 в МОД, а ребро 3-6 — в очередь с приоритетами. Ребро 1-3 помечается как непригодное.
- Удаляет из очереди с приоритетами непригодные ребра 1-3, 1-5 и 2-7.
- Добавляет вершину 5 и ребро 5-7 в МОД, а ребро 4-5 — в очередь с приоритетами. Ребро 1-5 помечается как непригодное.
- Добавляет вершину 4 и ребро 4-5 в МОД, а ребро 6-4 — в очередь с приоритетами. Ребра 4-7 и 0-4 помечаются как непригодные.
- Удаляет из очереди с приоритетами непригодные ребра 1-2, 4-7 и 0-4.
- Добавляет вершину 6 и ребро 6-2 в МОД. Остальные ребра, смежные с 6, помечаются как непригодные.

После добавления V вершин (и $V-1$ ребер) построение МОД завершено. Остальные ребра в очереди с приоритетами помечены как непригодные, поэтому просматривать их не нужно.

Реализация

После такой тщательной подготовки реализовать алгоритм Прима уже несложно — см. реализацию LazyPrimMST в листинге 4.3.5. Как в случае реализаций поиска в глубину и поиска в ширину в двух предыдущих разделах, она вычисляет МОД в конструкторе, чтобы клиентские методы могли получать свойства МОД с помощью методов запросов. В реализации используется приватный метод visit(), который помещает вершину в дерево, помечает ее как посещенную, а затем заносит в очередь с приоритетами все пригодные инцидентные этой вершине ребра — так гарантируется, что очередь с приоритетами содержит перекрестные ребра из вершин в дереве в вершины, не принадлежащие дереву (возможно, некоторые из них окажутся непригодными).

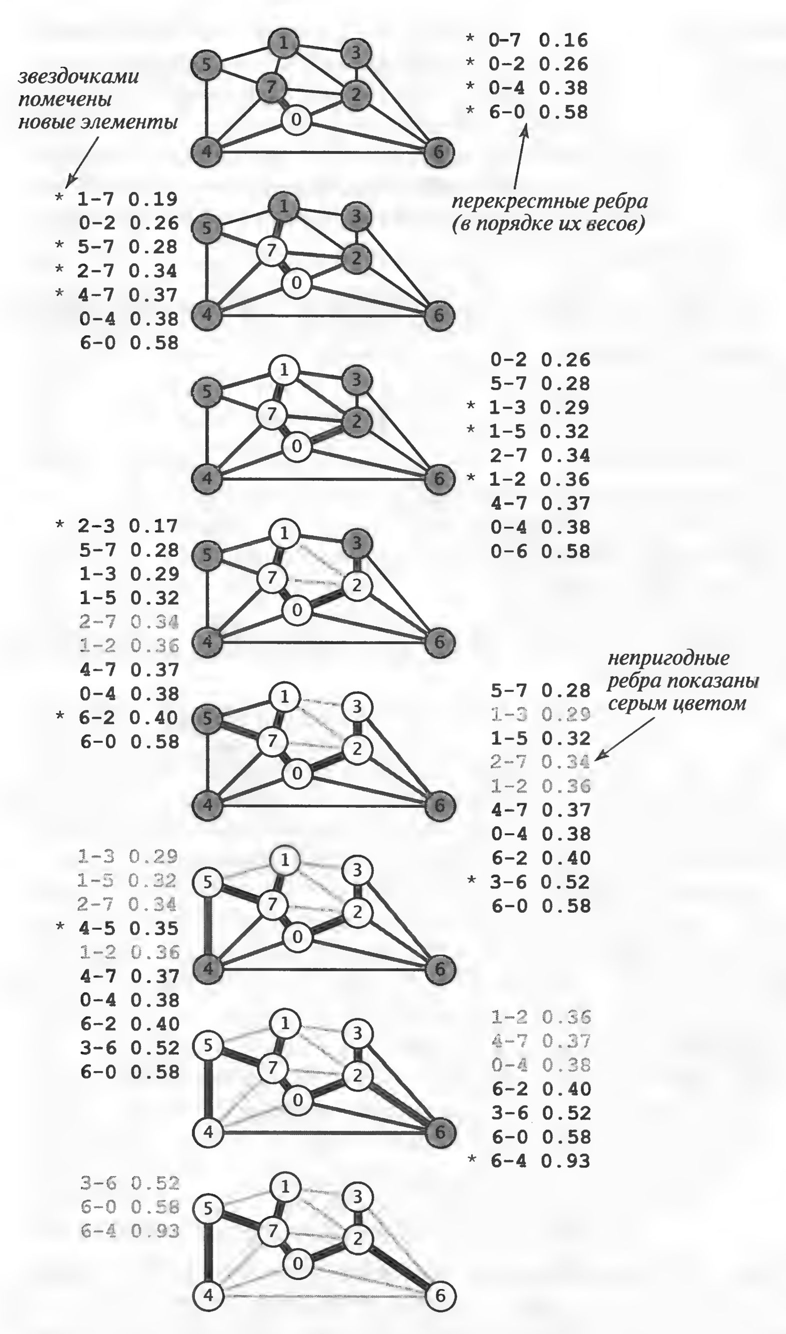


Рис. 4.3.14. *Трассировка алгоритма Прима ("ленивый" вариант)*

Внутренний цикл представляет собой программное выражение одного предложения с описанием алгоритма: ребро выбирается из очереди с приоритетами и (если оно пригодно) добавляется в дерево; кроме того, в дерево заносится новая вершина, в которую ведет это ребро, и изменяется множество перекрестных ребер с помощью вызова `visit()` с этой вершиной в качестве аргумента. Метод `weight()` требует перебора всех ребер дерева с добавлением весов ребер ("ленивый" способ) или хранит текущий общий вес в переменной экземпляров ("энергичный" способ); он оставлен на самостоятельную проработку в упражнении 4.3.31.

Листинг 4.3.5. "Ленивый" вариант алгоритма Прима для нахождения МОД

```
public class LazyPrimMST
{
   private boolean[] marked;          // вершины МОД
   private Queue<Edge> mst;           // ребра МОД
   private MinPQ<Edge> pq;            // перекрестные (и непригодные) ребра

   public LazyPrimMST(EdgeWeightedGraph G)
   {
      pq = new MinPQ<Edge>();
      marked = new boolean[G.V()];
      mst = new Queue<Edge>();

      visit(G, 0);   // предполагается, что G связный (см. упражнение 4.3.22)
      while (!pq.isEmpty())
      {
         Edge e = pq.delMin();                      // Извлечение из pq ребра
         int v = e.either(), w = e.other(v);        //    с минимальным весом.
         if (marked[v] && marked[w]) continue;      // Пропуск, если оно недопустимо.
         mst.enqueue(e);                            // Добавление ребра в дерево.
         if (!marked[v]) visit(G, v);               // Добавление вершины в дерево
         if (!marked[w]) visit(G, w);               //   (или v, или w).
      }
   }

   private void visit(EdgeWeightedGraph G, int v)
   {  // Помечает вершину v и заносит в pq все ребра из v в непомеченные вершины.
      marked[v] = true;
      for (Edge e : G.adj(v))
         if (!marked[e.other(v)]) pq.insert(e);
   }

   public Iterable<Edge> edges()
   {  return mst;  }

   public double weight()   // См. упражнение 4.3.31.
}
```

В этой реализации алгоритма Прима используется очередь с приоритетами для хранения перекрестных ребер, индексированные вершинами массивы для пометок вершин дерева и очередь для хранения ребер МОД. Здесь реализован "ленивый" способ, где непригодные ребра остаются в очереди с приоритетами.

Время выполнения

Насколько быстро работает алгоритм Прима? На этот вопрос нетрудно ответить, поскольку мы вооружены знанием о характеристиках очередей с приоритетами.

Утверждение Н. Для вычисления МОД связного графа с E взвешенными ребрами и V вершинами "ленивому" варианту алгоритма Прима требуется объем памяти, пропорциональный E, и время, пропорциональное $E \log E$ (в худшем случае).

Доказательство. Узким местом алгоритма является количество сравнений весов ребер в методах очереди с приоритетами `insert()` и `delMin()`. Количество ребер в очереди с приоритетами не превышает E, откуда получается верхняя граница объема памяти. В худшем случае стоимость вставки равна $\sim\lg E$, а стоимость извлечения минимального элемента равна $\sim 2\lg E$ (см. утверждение О из главы 2). Граница для времени выполнения следует из того, что вставляются не более E ребер, и удаляются не более E ребер.

В практических ситуациях верхняя граница времени выполнения весьма завышена, т.к. обычно количество ребер в очереди с приоритетами значительно меньше E. Существование такого простого, эффективного и полезного алгоритма для подобной сложной задачи — примечательный факт. А теперь мы кратко рассмотрим некоторые усовершенствования. Как обычно, детальное воплощение таких усовершенствований в приложениях, где важна производительность, должны выполнять эксперты.

"Энергичный" вариант алгоритма Прима

В качестве усовершенствования кода `LazyPrimMST` можно удалять непригодные ребра из очереди с приоритетами, чтобы в ней находились *только* перекрестные ребра между вершинами из дерева и вершинами не из дерева. Но можно устранить даже больше ребер. Ведь нас интересует только *минимальное* ребро из каждой вершины извне дерева в вершину в дереве. Добавление вершины `v` в дерево может повлиять на каждую вершину `w` не из дерева только одним способом: вершина `w` становится ближе к дереву, чем до этого (рис. 4.3.15). Ведь не нужно хранить в очереди с приоритетами *все* ребра от `w` к вершинам в дереве — достаточно отслеживать ребро с минимальным весом, а при добавлении `v` в дерево изменять этот минимум (т.к. ребро `v-w` имеет меньший вес) при обработке ребер в списке смежности `v`. То есть в очереди с приоритетами можно хранить лишь *одно* ребро от каждой вершины `w` вне дерева — кратчайшее ребро, соединяющее ее с деревом. Любое более длинное ребро, соединяющее `w` с деревом, однажды все равно станет непригодным, и его не стоит хранить в очереди с приоритетами.

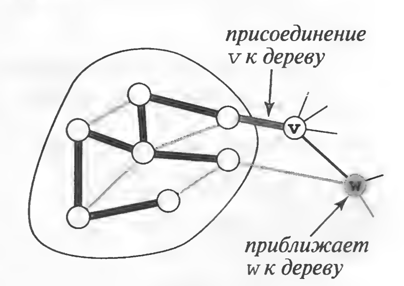


Рис. 4.3.15. "Энергичный" вариант алгоритма Прима

Класс `PrimMST` (алгоритм 4.7 в листинге 4.3.6) реализует алгоритм Прима с помощью нашего типа данных для индексной очереди с приоритетами из раздела 2.4 (см. листинг 2.4.8). По сравнению с реализацией `LazyPrimMST` в нем две структуры данных `marked[]` и `mst[]` заменены двумя массивами, которые индексированы вершинами, `edgeTo[]` и `distTo[]` со следующими свойствами:

- Если v не принадлежит дереву, но существует хотя бы одно ребро, соединяющее его с деревом, то edgeTo[v] — кратчайшее ребро, соединяющее v с деревом, а distTo[v] — вес этого ребра.
- Все подобные вершины v хранятся в индексной очереди с приоритетами, и индекс v связан с весом ребра edgeTo[v].

Листинг 4.3.6. Алгоритм 4.7. "Энергичный" вариант алгоритма Прима для нахождения МОД

```
public class  PrimMST
{
  private Edge[] edgeTo;              // кратчайшее ребро из вершины дерева
  private double[] distTo;            // distTo[w] = edgeTo[w].weight()
  private boolean[] marked;           // true, если v включена в дерево
  private IndexMinPQ<Double> pq;      // пригодные перекрестные ребра

  public PrimMST(EdgeWeightedGraph G)
  {
    edgeTo = new Edge[G.V()];
    distTo = new double[G.V()];
    marked = new boolean[G.V()];
    for (int v = 0; v < G.V(); v++)
      distTo[v] = Double.POSITIVE_INFINITY;
    pq = new IndexMinPQ<Double>(G.V());

    distTo[0] = 0.0;
    pq.insert(0, 0.0);                // Вначале в pq заносится 0 с весом 0.
    while (!pq.isEmpty())
      visit(G, pq.delMin());          // Добавление в дерево ближайшей вершины.
  }

  private void visit(EdgeWeightedGraph G, int v)
  { // Добавление v в дерево и изменение структур данных.
    marked[v] = true;
    for (Edge e : G.adj(v))
    {
      int w = e.other(v);
      if (marked[w]) continue;        // v-w непригодно.
      if (e.weight() < distTo[w])
      { // Ребро e — новое лучшее соединение дерева с w.
        edgeTo[w] = e;
        distTo[w] = e.weight();
        if (pq.contains(w)) pq.change(w, distTo[w]);
        else                pq.insert(w, distTo[w]);
      }
    }
  }

  public Iterable<Edge> edges()       // См. упражнение 4.3.21.
  public double weight()              // См. упражнение 4.3.31.
}
```

В этой реализации алгоритма Прима пригодные перекрестные ребра хранятся в индексной очереди с приоритетами.

Основные следствия из этих свойств: *минимальный ключ в очереди с приоритетами — это вес перекрестного ребра с минимальным весом, и следующим шагом нужно добавить в дерево связанную с ним вершину* v.

Массив marked[] уже не нужен, т.к. условие !marked[w] эквивалентно условию бесконечности значения distTo[w] (и нулевого значения edgeTo[w]). При работе со структурами данных алгоритм выбирает из очереди с приоритетами ребро v, а потом проверяет каждое ребро v-w из ее списка смежности. Если вершина w помечена, ребро непригодно. А если она не находится в очереди с приоритетами или ее вес меньше текущего известного edgeTo[w], то код изменяет содержимое структур данных, чтобы ребро v-w стало лучшим известным путем соединения v с деревом.

На рис. 4.3.16 показана трассировка работы PrimMST с нашим небольшим демонстрационным графом tinyEWG.txt. Содержимое массивов edgeTo[] и distTo[] приведено после добавления в МОД каждой вершины, причем черным цветом выделены вершины МОД, светло-серым — вершины, не принадлежащие МОД, черным — ребра МОД и жирным черным — пары индекс/значение из очереди с приоритетами. На диаграммах кратчайшее ребро, соединяющее каждую вершину извне МОД с вершиной МОД, показано темно-серым цветом. Алгоритм добавляет ребра в МОД в том же порядке, что и "ленивый" вариант, и разница лишь в операциях очереди с приоритетами. Он строит МОД следующим образом.

- Добавляет вершину 0 в МОД, а все ребра из ее списка смежности — в очередь с приоритетами, т.к. каждое такое ребро — это наилучшее (единственно) известное соединение между вершиной из дерева и вершиной не из дерева.
- Добавляет в МОД вершину 7 и ребро 0-7, а ребра 1-7 и 5-7 — в очередь с приоритетами. Ребра 4-7 и 2-7 не изменяют очередь с приоритетами, т.к. их веса не меньше весов известных соединений между МОД и вершинами 4 и 2 соответственно.
- Добавляет в МОД вершину 1 и ребро 1-7, а ребро 1-3 — в очередь с приоритетами.
- Добавляет в МОД вершину 2 и ребро 0-2, заменяет ребро 0-6 на 2-6 (более коротким ребром из вершины дерева в 6) и ребро 1-3 на 2-3 (более коротким ребром из вершины дерева в 3).
- Добавляет в МОД вершину 3 и ребро 2-3.
- Удаляет из очереди с приоритетами непригодные ребра 1-3, 1-5 и 2-7.
- Добавляет в МОД вершину 5 и ребро 5-7, и заменяет ребро 0-4 на 4-5 (более короткое ребро из вершины дерева в 4).
- Добавляет в МОД вершину 4 и ребро 4-5.
- Добавляет в МОД вершину 6 и ребро 6-2.

После добавления $V-1$ ребер построение МОД завершено, а очередь с приоритетами пуста.

Рассуждение, почти идентичное доказательству утверждения Н, доказывает, что "энергичный" вариант алгоритма Прима находит МОД в связном графе с взвешенными ребрами за время, пропорциональное $E \log V$, и задействует объем памяти, пропорциональный V (см. утверждение О). Для больших разреженных графов, которые обычно встречаются на практике, нет асимптотической разницы в граничных значениях времени (поскольку для разреженных графов $\lg E \sim \lg V$), а граничный объем памяти улучшен в постоянное (хотя и значительное) количество раз.

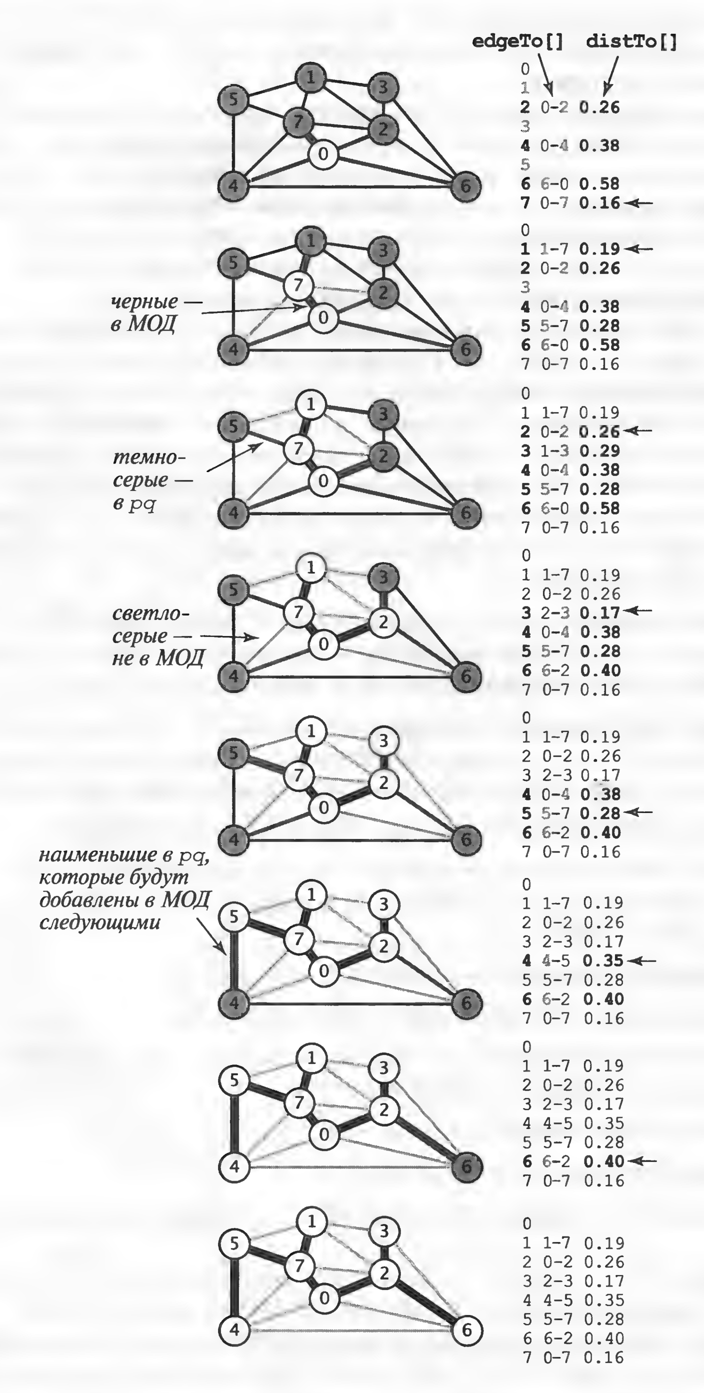


Рис. 4.3.16. Трассировка алгоритма Прима ("энергичный" вариант)

Более подробный анализ и эмпирические исследования лучше оставить специалистам, которые работают с приложениями, где важна производительность, т.к. в них требуется учитывать множество факторов: реализации MinPQ и IndexMinPQ, представление графа, свойства модели графа для конкретного приложения и т.д. Как обычно, такие усовершенствования требуют тщательного анализа, ведь повышение сложности кода оправдано только в тех приложениях, где коэффициент повышения производительности значителен, иначе оно может привести даже к ухудшению работы в современных сложных системах.

Утверждение О. Для вычисления МОД связного графа с E взвешенными ребрами и V вершинами "энергичному" варианту алгоритма Прима требуется объем памяти, пропорциональный V, и время, пропорциональное $E \log V$ (в худшем случае).

Доказательство. Количество ребер в очереди с приоритетами не превышает V, и задействованы три индексированных вершинами массива — отсюда получается верхняя граница объема памяти. Алгоритм выполняет V операций *вставить*, V операций *извлечь наименьший* и (в худшем случае) E операций *изменить приоритет*. Эти счетчики, а также тот факт, что наша пирамидальная реализация индексной очереди с приоритетами выполняет все эти операции за время, пропорциональное $\log V$ (см. табл. 2.4.4), и определяют граничное значение времени.

На рис. 4.3.17 показаны этапы работы алгоритма Прима на нашем евклидовом графе mediumEWG.txt с 250 вершинами. Это завораживающий динамический процесс (см. также упражнение 4.3.27). Чаще всего дерево разрастается за счет добавления вершины к только что добавленной вершине. Когда процесс упирается в место, где рядом уже нет вершин вне дерева, рост начинается с другой части дерева.

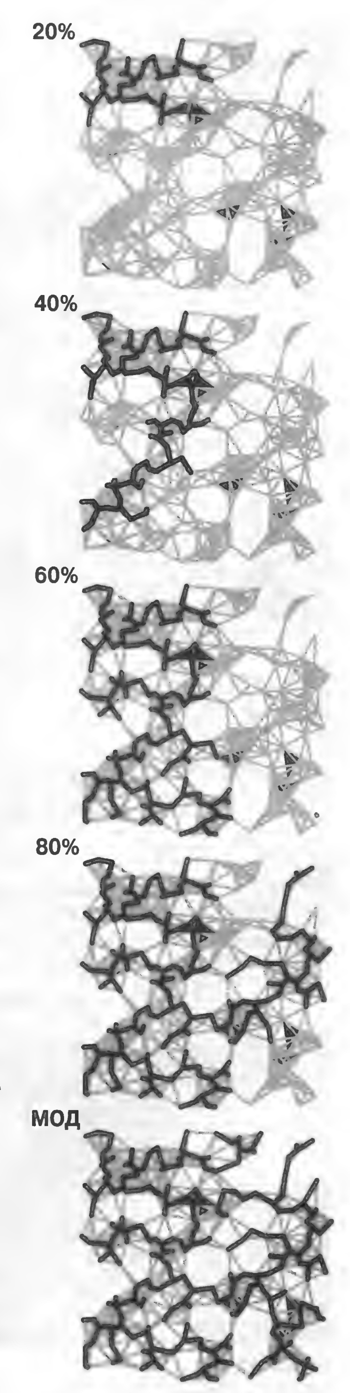


Рис. 4.3.17. Этапы работы алгоритма Прима (250 вершин)

Алгоритм Крускала

Второй алгоритм построения МОД, который мы подробно рассмотрим, обрабатывает ребра в порядке возрастания их весов и присоединяет к МОД (на рисунке выделено черным цветом) каждое ребро, которое не замыкает цикл с уже добавленными ребрами; он завершает свою работу после добавления $V-1$ ребер. Черные ребра образуют лес деревьев, который постепенно срастается в единое дерево МОД. Этот метод называется *алгоритмом Крускала* (Kruskal).

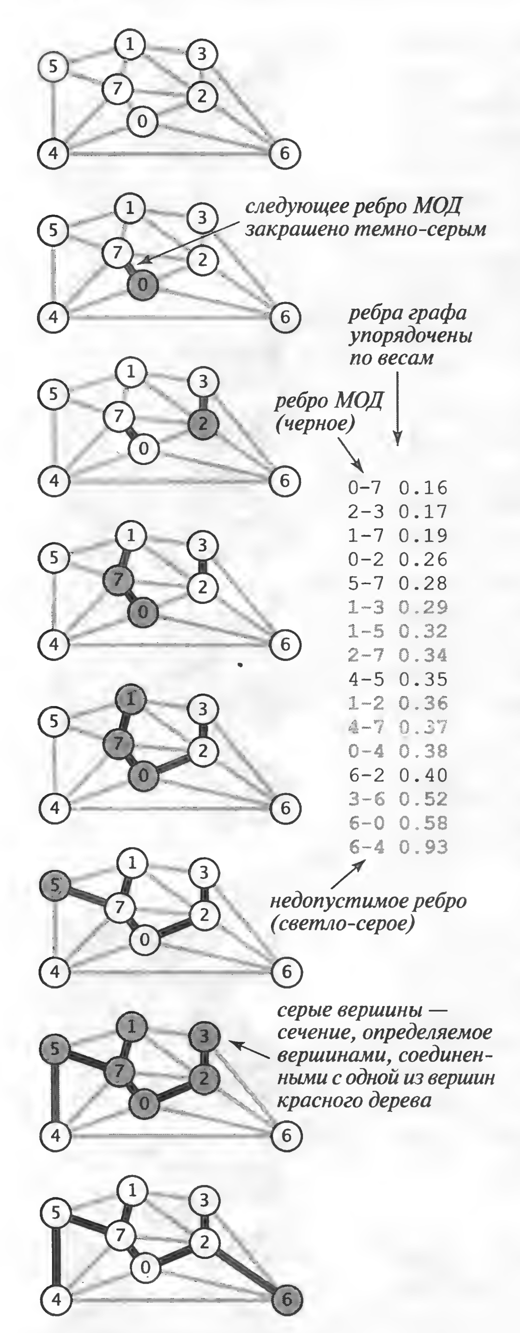


Рис. 4.3.18. Трассировка алгоритма Крускала

Утверждение П. Алгоритм Крускала вычисляет МОД для любого связного графа с взвешенными ребрами.

Доказательство. Непосредственно следует из утверждения Л. Если следующее рассматриваемое ребро не образует цикла с черными ребрами, оно пересекает сечение, которое определяется множеством вершин, соединенных с одной из вершин ребрами дерева (и его дополнением). Поскольку ребро не образует цикла, это единственное встреченное к данному моменту перекрестное ребро, а поскольку мы рассматриваем ребра в порядке возрастания веса, это перекрестное ребро минимального веса. Поэтому алгоритм последовательно выбирает перекрестные ребра минимального веса, в соответствии с "жадным" алгоритмом.

Алгоритм Прима строит МОД, находя на каждом шаге новое ребро и присоединяя его к единому растущему дереву. Алгоритм Крускала тоже строит МОД, добавляя ребра по одному, но он находит ребро, которое соединяет два дерева в лесе растущих деревьев. Построение начинается с вырожденного леса, который состоит из V деревьев из одной вершины, и дальше выполняются объединения двух деревьев (с помощью минимально возможного ребра), пока не останется только одно дерево — это и будет МОД.

На рис. 4.3.18 приведен пример пошагового выполнения действий алгоритма Крускала для графа `tinyEWG.txt`. Пять ребер с наименьшими весами выбраны из графа в МОД, потом ребра 1-3, 1-5 и 2-7 признаны непригодными, ребро 4-5 вставлено в МОД, ребра 1-2, 4-7 и 0-4 опять признаны непригодными и, наконец, ребро 6-2 добавлено в МОД.

Алгоритм Крускала также нетрудно реализовать — пользуясь базовыми алгоритмическими средствами, которые уже рассмотрены в данной книге. Очередь с приоритетами (раздел 2.4) используется для выборки ребер в порядке возрастания весов, структура данных объединения-поиска (раздел 1.5) позволяет находить ребра, замыкающие циклы, а очередь (раздел 1.3) применяется для сбора ребер МОД. Алгоритм 4.8 представляет собой реализацию на основе всех этих средств.

Учтите, что накопление ребер МОД в объекте Queue означает, что когда клиент перебирает ребра, он получает их в порядке возрастания их весов. Такой перебор нужен в методе weight() для суммирования весов ребер (или хранения текущей суммы в переменной экземпляров), который оставлен на самостоятельную проработку (см. упражнение 4.3.31).

Листинг 4.3.7. Алгоритм 4.8. Алгоритм Крускала для нахождения МОД

```
public class  KruskalMST
{
  private Queue<Edge> mst;

  public KruskalMST(EdgeWeightedGraph G)
  {
    mst = new Queue<Edge>();
    MinPQ<Edge> pq = new MinPQ<Edge>(G.edges());
    UF uf = new UF(G.V());

    while (!pq.isEmpty() && mst.size() < G.V()-1)
    {
      Edge e = pq.delMin();              // Извлечение из pq ребра с минимальным
      int v = e.either(), w = e.other(v);   //    весом и его вершин.
      if (uf.connected(v, w)) continue;     // Непригодные ребра игнорируются.
      uf.union(v, w);                       // Слияние компонентов.
      mst.enqueue(e);                       // Добавление ребра в mst.
    }
  }

  public Iterable<Edge> edges()
  {  return mst;  }

  public double weight()                    // См. упражнение 4.3.31.
}
```

В этой реализации алгоритма Крускала используется очередь с приоритетами для хранения еще не просмотренных ребер и структура данных объединения-поиска для выявления непригодных ребер. Ребра МОД возвращаются клиенту в порядке возрастания их весов. Метод weight() оставлен в качестве упражнения.

```
% java KruskalMST tinyEWG.txt
0-7 0.16
2-3 0.17
1-7 0.19
0-2 0.26
5-7 0.28
4-5 0.35
6-2 0.40
1.81
```

Анализ времени выполнения алгоритма Крускала не представляет сложности, т.к. известны значения времени выполнения всех его базовых операций.

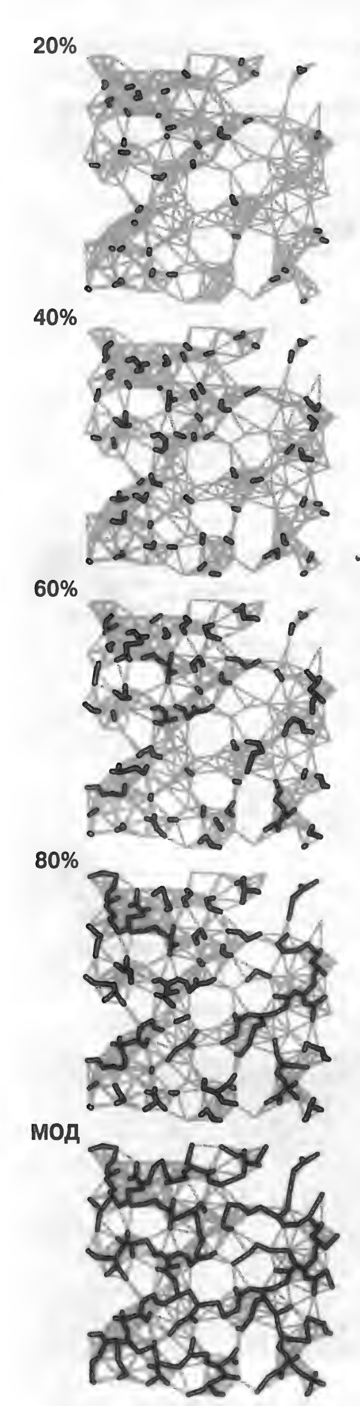


Рис. 4.3.19. Этапы работы алгоритма Крускала (250 вершин)

Утверждение О (продолжение). Для вычисления МОД связного графа с E взвешенными ребрами и V вершинами алгоритму Крускала требуется объем памяти, пропорциональный E, и время, пропорциональное $E \log E$ (в худшем случае).

Доказательство. В реализации применяется конструктор очереди с приоритетами, который заносит в очередь все ребра, выполнив не более E сравнений (см. раздел 2.4). После построения очереди с приоритетами повторяется такое же рассуждение, как и при анализе алгоритма Прима. Количество ребер в очереди с приоритетами не превышает E — это дает верхнюю оценку объема памяти, а на одну операцию требуется не более $2\lg E$ сравнений, что дает границу для времени выполнения. Алгоритм Крускала выполняет также до E операций `find()` и до V операций `union()`, но их трудоемкость не повышает порядок роста $E \log E$ для общего времени выполнения (см. раздел 1.5).

Как и в случае алгоритма Прима, оценка стоимости чересчур осторожна, т.к. алгоритм завершает работу после нахождения $V-1$ ребер МОД. Реальный порядок роста стоимости равен $E+ E_0 \log E$, где E_0 — количество ребер, вес которых меньше веса максимального ребра МОД. Несмотря на это преимущество, алгоритм Крускала обычно работает медленнее алгоритма Прима, т.к. для каждого ребра ему приходится выполнять операцию `connected()` — кроме операций в очереди с приоритетами, которые выполняют оба алгоритма для каждого обрабатываемого ребра (см. упражнение 4.3.39).

На рис. 4.3.19 показаны динамические характеристики алгоритма Крускала на примере более крупного файла `mediumEWG.txt`. Наглядно видно, что ребра добавляются в лес в порядке возрастания их длины.

Перспектива

Задача нахождения МОД — одна из наиболее изученных задач из описанных в данной книге. Принципиальные методы ее решения были изобретены задолго до разработки современных структур данных и современных способов анализа производительности алгоритмов — еще в те времена, когда нахождение МОД в графе со, скажем, несколькими тысячами ребер было утомительным делом. Изученные нами алгоритмы нахождения МОД отличаются от тех первых в основном использованием и реализацией современных алгоритмов и структур данных для базовых задач, которые (вместе с современной вычислительной мощью) позволяют вычислять минимальные остовные деревья с миллионами или даже миллиардами ребер.

Исторические сведения

Реализация вычисления МОД для насыщенных графов (см. упражнение 4.3.29) была впервые предложена Примом (R. Prim) в 1961 г. и вскоре независимо Дейкстрой (E.W. Dijkstra). Обычно ее называют *алгоритмом Прима*, хотя предложенный Дейкстрой вариант был более общим. Но основная идея была представлена еще и Ярником (V. Jarnik), поэтому некоторые авторы называют данный метод *алгоритмом Ярника*, подчеркивая, что Прим (и Дейкстра) просто нашли эффективную реализацию алгоритма для насыщенных графов. Когда в начале 1970-х годов начали применяться АТД очередей с приоритетами, их использование для нахождения МОД было сразу очевидным, и факт, что минимальные остовные деревья для разреженных графов можно вычислять за время, пропорциональное $E \log E$, стал общепризнанным без какого-то конкретного автора. В 1984 г. Фредман (M.L. Fredman) и Тарьян (R.E. Tarjan) разработали структуру данных *пирамида Фибоначчи*, которая улучшает теоретическую границу порядка роста для времени выполнения алгоритма Прима до $E + V \log V$. Крускал (J. Kruskal) опубликовал свой алгоритм в 1956 г., но и в этом случае соответствующие реализации АТД не были хорошо изучены еще много лет. Любопытно, что в статье Крускала был описан и вариант алгоритма Прима, а в 1926 г. (!) в статье Борувки (O. Boruvka) были упомянуты оба эти подхода. В статье Борувки описывается приложение для распределения мощности и еще один метод, который легко реализовать с помощью современных структур данных (см. упражнения 4.3.43 и 4.3.44). Этот метод был заново открыт Соллином (M. Sollin) в 1961 г.; позднее он привлек внимание как основа для параллельных алгоритмов нахождения МОД с эффективной асимптотической производительностью.

Таблица 4.3.2. Характеристики производительности алгоритмов нахождения МОД

Алгоритм	Порядок роста в худшем случае для V вершин и E ребер	
	Память	Время
"Ленивый" Прима	E	$E \log E$
"Энергичный" Прима	V	$E \log V$
Крускала	E	$E \log E$
Фредмана-Тарьяна	V	$E + V \log V$
Шазеля	V	Очень близко к E, но не точно E
невозможно?	V	E?

Линейный алгоритм?

С одной стороны, нет никаких теоретических результатов, отрицающих существование алгоритма МОД, который гарантированно выполняется за линейное время для всех графов. С другой стороны, разработка алгоритмов для вычисления МОД разреженных графов за линейное время так и не увенчалась успехом. С 1970-х годов применимость абстракции объединения-поиска в алгоритме Крускала и применимость абстракции очереди с приоритетами в алгоритме Прима являются основными мотивами для многих исследователей, которые пытаются найти лучшие реализации этих АТД. Многие исследователи стараются найти эффективные реализации очереди с приоритетами, как осно-

ву для поиска эффективных алгоритмов МОД для разреженных графов. Многие другие исследователи обратились к различным вариантам алгоритма Борувки, пытаясь на их основе найти почти линейные алгоритмы нахождения МОД для разреженных графов. Эти поиски все еще могут привести к обнаружению практического линейного алгоритма МОД и даже показали существование рандомизированного линейного алгоритма. Кроме того, исследователи довольно близко подобрались к линейной цели: в 1997 г. Шазель (B. Chazelle) опубликовал алгоритм, который невозможно отличить от линейного алгоритма в любой разумной практической ситуации (хотя он, возможно, не является линейным). Однако этот алгоритм настолько сложен, что никто не захочет связываться с ним на практике.

Алгоритмы, найденные в этих исследованиях, обычно весьма сложны, но упрощенные версии некоторых из них могут оказаться вполне пригодными на практике. А пока мы можем пользоваться базовыми алгоритмами, которые были рассмотрены здесь для вычисления МОД за линейное время в большинстве практических ситуаций — возможно, за счет дополнительного коэффициента $\log V$ для некоторых разреженных графов.

В общем, задачу нахождения МОД для практических целей можно считать "решенной". Для большинства графов стоимость нахождения МОД лишь ненамного превышает стоимость извлечения ребер графа. Это правило нарушается только для огромных и очень разреженных графов, но даже в этом случае полученный выигрыш в производительности по сравнению с хорошо известными алгоритмами не превышает 10 раз. Эти выводы подтверждаются для многих моделей графов, а практики используют алгоритмы Прима и Крускала для нахождения МОД уже десятки лет.

Вопросы и ответы

Вопрос. **Работают ли алгоритмы Прима и Крускала для *ориентированных* графов?**

Ответ. Нет. Это более трудная задача обработки графов, которая называется задачей поиска *древовидной структуры минимальной стоимости*.

Упражнения

4.3.1. Докажите, что если изменить все веса ребер, прибавив к ним положительную константу или умножив их на положительную константу, то МОД при этом не изменится.

4.3.2. Нарисуйте все минимальные остовные деревья для графа, приведенного на рис. 4.3.20 (веса всех ребер одинаковы).

Рис. 4.3.20. *Граф для упражнения 4.3.2*

4.3.3. Покажите, что если веса всех ребер графа различны, то его МОД уникально.

4.3.4. Проанализируйте утверждение, что граф с взвешенными ребрами имеет уникальное МОД *только* тогда, когда веса его ребер различны. Приведите доказательство или контрпример.

4.3.5. Покажите, что "жадный" алгоритм работает правильно и при наличии ребер с одинаковыми весами.

4.3.6. Приведите МОД для взвешенного графа, полученного из файла `tinyEWG.txt` (рис. 4.3.1) удалением вершины 7.

4.3.7. Как найти *максимальное* остовное дерева графа с взвешенными ребрами?

4.3.8. Докажите следующее утверждение, которое называется *свойством цикла*: при наличии любого цикла в графе с взвешенными ребрами (все веса различны) ребро с максимальным весом в цикле не принадлежит МОД этого графа.

4.3.9. Реализуйте конструктор для класса `EdgeWeightedGraph`, который читает граф из входного потока. Для этого возьмите и измените конструктор из класса `Graph` (листинг 4.1.1).

4.3.10. Разработайте реализацию `EdgeWeightedGraph` для насыщенных графов с представлением матрицей смежности (двумерный массив весов). Отбрасывайте параллельные ребра.

4.3.11. Определите объем памяти, необходимой реализации `EdgeWeightedGraph` для представления графа с V вершинами и E ребрами. Используйте модель стоимости памяти из раздела 1.4.

4.3.12. Пусть в графе все ребра имеют разные веса. Обязательно ли самое короткое ребро принадлежит МОД? Может ли принадлежать МОД самое длинное ребро? Обязательно ли принадлежит МОД ребро с наименьшим весом на каждом цикле? Обоснуйте свои ответы на каждый вопрос или приведите контрпримеры.

4.3.13. Приведите контрпример, который показывает, что следующая стратегия не обязательно находит МОД: начинаем с любой вершины как с МОД из единственной вершины, а потом добавляем к ней $V-1$ ребер, всегда выбирая следующее ребро с минимальным весом, инцидентное вершине, которая позже всего добавлена в МОД.

4.3.14. Пусть имеется МОД для графа G с взвешенными ребрами, и после этого из G удалено ребро, которое не разъединяет G. Опишите, как можно найти МОД нового графа за время, пропорциональное E.

4.3.15. Пусть имеется МОД для графа G с взвешенными ребрами, и в него вставлено новое ребро e. Опишите, как можно найти МОД нового графа за время, пропорциональное V.

4.3.16. Пусть имеется МОД для графа G с взвешенными ребрами, и в него вставлено новое ребро e. Напишите программу, которая определяет диапазон весов, для которых e входит в МОД.

4.3.17. Реализуйте метод `toString()` для класса `EdgeWeightedGraph`.

4.3.18. Приведите трассировки, которые демонстрируют процесс вычисления МОД в графе, определенном в упражнении 4.3.6, с помощью "ленивого" варианта алгоритма Прима, "энергичного" варианта алгоритма Прима и алгоритма Крускала.

4.3.19. Пусть используется реализация очереди с приоритетами на основе упорядоченного списка. Каков порядок роста времени выполнения в худшем случае алгоритма Прима и алгоритма Крускала для графов с V вершинами и E ребрами? Когда стоит применять этот метод и стоит ли вообще? Обоснуйте свой ответ.

4.3.20. Правда ли, что в любой момент выполнения алгоритма Крускала каждая вершина ближе к некоторой вершине в своем поддереве, чем к любой вершине извне этого поддерева? Обоснуйте свой ответ.

4.3.21. Напишите реализацию метода `edges()` для класса `PrimMST` (листинг 4.3.6).

Решение:

```
public Iterable<Edge> edges()
{
   Bag<Edge> mst = new Bag<Edge>();
   for (int v = 1; v < edgeTo.length; v++)
      mst.add(edgeTo[v]);
   return mst;
}
```

Творческие задачи

4.3.22. *Минимальный остовный лес.* Напишите варианты алгоритмов Прима и Крускала, которые вычисляют минимальный остовный *лес* графа с взвешенными ребрами, который не обязательно связен. Используйте API для связных компонентов (рис. 4.1.27) и найдите минимальные остовные деревья в каждом компоненте.

4.3.23. *Алгоритм Высоцкого.* Разработайте реализацию вычисления МОД на основе повторения свойства цикла (см. упражнение 4.3.8): ребра поочередно добавляются в предполагаемое дерево, а если при этом образуется цикл, то из него удаляется ребро с максимальным весом. *Примечание*: этот метод не так привлекателен, как другие, из-за относительной сложности использования структуры данных, которая поддерживает эффективную реализацию операции *удалить из цикла ребро с максимальным весом*.

4.3.24. *Алгоритм обратного удаления.* Разработайте реализацию, которая вычисляет МОД следующим образом: начинаем с графа, содержащего все ребра. Затем многократно перебираем ребра в порядке убывания их весов. Для каждого ребра проверяем, разъединит ли граф удаление этого ребра; если нет, то удаляем его. Докажите, что этот алгоритм действительно оставляет МОД. Каков порядок роста для количества сравнений весов ребер, которые выполняет ваша реализация?

4.3.25. *Генератор худших случаев.* Разработайте приемлемый генератор графов с V вершинами и E взвешенными ребрами, для которых время выполнения "ленивого" варианта алгоритма Прима нелинейно. Сделайте то же самое для "энергичного" варианта.

4.3.26. *Критические ребра.* Ребро МОД, удаление которого из графа приводит к увеличению веса МОД, называется *критическим ребром*. Покажите, как найти все критические ребра в графе за время, пропорциональное $E \log E$. *Примечание*: в этом вопросе веса некоторых ребер должны быть совпадать (иначе все ребра МОД будут критическими).

4.3.27. *Анимации.* Напишите клиентскую программу, которая выполняет динамические графические анимации для алгоритмов нахождения МОД. Выполните эту программу для графа `mediumEWG.txt`, и получите изображения, подобные показанным на рис. 4.3.16 и 4.3.18.

4.3.28. *Структуры данных, экономящие память.* Разработайте реализацию "ленивого" варианта алгоритма Прима, который экономит память, используя для `EdgeWeightedGraph` и `MinPQ` низкоуровневые структуры данных вместо объектов `Bag` и `Edge`. Оцените объем сэкономленной памяти в виде функции от V и E на основе модели стоимости памяти из раздела 1.4 (см. упражнение 4.3.11).

4.3.29. *Насыщенные графы*. Разработайте реализацию алгоритма Прима на основе “энергичного” подхода (но без очереди с приоритетами), которая вычисляет МОД с помощью V^2 сравнений весов ребер.

4.3.30. *Евклидовы взвешенные графы*. На основе решения упражнения 4.1.37 получите API `EuclideanEdgeWeightedGraph` для графов, вершины которых представляют собой точки на плоскости, чтобы работать с графическими представлениями графов.

4.3.31. *Веса МОД*. Разработайте реализации метода `weight()` для классов `LazyPrimMST`, `PrimMST` и `KruskalMST`, используя “*ленивую*” стратегию, которая перебирает ребра МОД при вызове метода `weight()` из клиента. Затем разработайте другие реализации на основе “*энергичной*” стратегии, которая отслеживает текущий общий вес при вычислении МОД.

4.3.32. *Заданное множество*. Пусть имеется связный граф G с взвешенными ребрами и заданное множество ребер S (не содержащее циклов). Опишите, как можно найти остовное дерево графа G с минимальным весом, которое содержит все ребра из S.

4.3.33. *Проверка*. Напишите клиент `check()` классов MST и `EdgeWeightedGraph`, который проверяет, что предложенное множество ребер на самом деле является МОД, используя *условия оптимальности сечений*, которое вытекает из утверждения К: множество ребер составляет МОД, если оно является остовным деревом, и каждое ребро имеет минимальный вес в сечении, определяемом удалением этого ребра из дерева. Каков порядок роста времени выполнения для этого метода?

Эксперименты

4.3.34. *Случайные разреженные графы с взвешенными ребрами*. Напишите генератор случайных разреженных графов с взвешенными ребрами, основанный на решении упражнения 4.1.41. Чтобы присваивать ребрам веса, определите АТД орграфа со случайными весами ребер и напишите две реализации: одна из которых генерирует веса с равномерным распределением, а другая — с гауссовым распределением. Напишите клиентские программы для генерации разреженных случайных графов с взвешенными ребрами для обоих распределений весов со специально подобранными значениями V и E, чтобы затем использовать их для выполнения эмпирических тестов на графах с различными распределениями весов ребер.

4.3.35. *Случайные евклидовы графы с взвешенными ребрами*. Измените решение упражнения 4.1.42, используя в качестве весов ребер расстояния между вершинами.

4.3.36. *Случайные графы на решетке с взвешенными ребрами*. Измените решение упражнения 4.1.43, используя в качестве весов ребер случайные числа от 0 до 1.

4.3.37. *Реальные графы с взвешенными ребрами*. Найдите в Интернете большой взвешенный граф — карту расстояний, схему телефонных соединений с их стоимостями или таблицу стоимостей авиарейсов. Напишите программу `RandomRealEdgeWeightedGraph`, которая строит взвешенный граф, выбирая из этого графа V случайных вершины и E взвешенных ребер, индуцированных этими вершинами.

Тестирование всех алгоритмов со всеми возможными параметрами на всех моделях графов выполнить нереально. Для каждой из приведенных ниже задач напишите клиент, решающий эту задачу, а затем выберите один из описанных выше генераторов, чтобы выполнять эксперименты для данной модели графов. Планируйте эксперименты обдуманно, возможно, на основе результатов предыдущих экспериментов. Напишите краткий анализ полученных результатов и выводы из этих результатов.

4.3.38. *Цена лени.* Эмпирически сравните производительность "ленивого" и "энергичного" вариантов алгоритма Прима, для различных видов графов.

4.3.39. *Сравнение алгоритмов Прима и Крускала.* Эмпирически сравните производительность "ленивого" и "энергичного" вариантов алгоритма Прима с алгоритмом Крускала.

4.3.40. *Снижение сопутствующих затрат.* Эмпирически определите эффект использования примитивных типов вместо значений типа `Edge` в реализации `EdgeWeightedGraph`, как описано в упражнении 4.3.28.

4.4. Кратчайшие пути

Пожалуй, наиболее наглядной и понятной задачей обработки графов является задача, с которой мы регулярно сталкиваемся при работе с картами или навигационной системой, чтобы найти путь из одного места в другое. Аналогия с графами здесь очевидна: вершины соответствуют перекресткам, ребра — дорогам, а веса ребер могут соответствовать, например, расстояниям или времени проезда. Возможность наличия дорог с односторонним движением означает, что необходимо рассматривать *орграфы* с взвешенными ребрами. В такой модели задачу сформулировать легко.

Найти путь наименьшей стоимости из одной вершины в другую.

Кроме подобных очевидных применений, модель кратчайших путей пригодна для решения целого спектра задач (см. табл. 4.4.1), часть которых на первый взгляд вообще никак не связана с обработкой графов. В качестве одного из таких примеров мы рассмотрим в конце раздела задачу *арбитража*.

Таблица 4.4.1. Типичные применения кратчайших путей

Область	Вершина	Ребро
Карта	Перекресток	Дорога
Сеть	Маршрутизатор	Соединение
Планирование	Работа	Ограничение предшествования
Арбитраж	Валюта	Курс обмена

Мы будем придерживаться общей модели и работать с *орграфами с взвешенными ребрами* (т.е. с сочетанием моделей из разделов 4.2 и 4.3, которое иногда называется *реберно-взвешенный орграф*). В разделе 4.2 мы хотели узнать, *возможно ли* попасть из одной вершины в другую; в данном разделе мы будем учитывать веса ребер, как это было в случае неориентированных графов с взвешенными ребрами в разделе 4.3. С каждым ориентированным путем в орграфе с взвешенными ребрами связывается *вес пути*, равный сумме весов ребер, которые входят в путь. Эта важная величина позволяет сформулировать такие задачи, как “найти ориентированный путь наименьшего веса из одной вершины в другую” — тема данного раздела. Пример такой задачи приведен на рис. 4.4.1.

Орграф с взвешенными ребрами

```
4->5  0.35
5->4  0.35
4->7  0.37
5->7  0.28
7->5  0.28
5->1  0.32
0->4  0.38
0->2  0.26
7->3  0.39
1->3  0.29
2->7  0.34
6->2  0.40
3->6  0.52
6->0  0.58
6->4  0.93
```


Кратчайший путь из 0 в 6

```
0->2  0.26
2->7  0.34
7->3  0.39
3->6  0.52
```

Рис. 4.4.1. Орграф с взвешенными ребрами и кратчайший путь

Определение. *Кратчайший путь* из вершины s в вершину t в орграфе с взвешенными ребрами — это такой ориентированный путь из s в t, что никакой другой такой путь не имеет меньший вес.

Так что в данном разделе мы изучим классические алгоритмы для решения следующей задачи.

Кратчайшие пути из одного источника. Для заданного орграфа с взвешенными ребрами и исходной вершины s необходимо отвечать на запросы вида *Существует ли ориентированный путь из* s *в указанную целевую вершину* t? Если да, то надо найти *кратчайший* такой путь (с минимальным суммарным весом).

Мы рассмотрим в этом разделе следующие темы.

- API и реализации для орграфов с взвешенными ребрами и API для поиска кратчайших путей из одного источника.
- Классический алгоритм Дейкстры для задачи с неотрицательными весами.
- Более быстрый алгоритм для ациклических орграфов с взвешенными ребрами (ориентированные ациклические графы с взвешенными ребрами), который работает даже в том случае, если веса ребер могут быть отрицательными.
- Классический алгоритм Беллмана-Форда для общего случая, когда возможно наличие циклов, веса ребер могут быть отрицательными, и нужен алгоритм для нахождения циклов с отрицательным весом и кратчайших путей в орграфах с взвешенными ребрами без таких циклов.

При изучении алгоритмов мы будем рассматривать и примеры их применения.

Свойства кратчайших путей

Базовое определение для задачи поиска кратчайших путей сформулировано исчерпывающе, но за его краткостью скрыто несколько моментов, которые стоит рассмотреть, прежде чем перейти к определению алгоритмов и соответствующих структур данных.



Рис. 4.4.2. *ДКП с 250 вершинами*

- *Пути являются ориентированными.* Кратчайший путь должен учитывать направления ребер.
- *Веса ребер не обязательно соответствуют расстояниям.* Визуальная наглядность может помочь в изучении алгоритмов, поэтому мы будем использовать примеры, где вершины представляют собой точки на плоскости, а веса ребер соответствуют геометрическим расстояниям между ними (как на рис. 4.4.1). Но веса могут представлять время, стоимость или еще какую-нибудь отвлеченную величину, которая совсем не обязательно пропорциональна расстоянию. Мы будем подчеркивать этот момент, используя смешанную терминологию: ищется *кратчайший* путь минимального *веса* или *стоимости.*
- *Не все вершины могут быть достижимыми.* Если вершина t не достижима из s, между ними вообще нет никакого пути — значит, не существует и кратчайший путь из s в t. Для простоты наш самый маленький пример обладает сильной связностью (каждая вершина достижима из любой другой вершины).
- *Отрицательные веса усложняют дело.* Вначале мы будем считать, что веса ребер положительны (или равны нулю). Неприятное влияние отрицательных весов — основная тема последней части данного раздела.

- *Кратчайшие пути обычно простые.* Наши алгоритмы игнорируют ребра нулевого веса, которые образуют циклы, поэтому они находят кратчайшие пути, не содержащие циклов.
- *Кратчайшие пути не обязательно уникальны.* Могут существовать несколько путей минимального веса из одной вершины в другую; нам достаточно найти любой из них.
- *Граф может содержать параллельные ребра и циклы.* В наборе параллельных ребер для нас важно лишь ребро с минимальным весом, и ни один кратчайший путь не содержит петель (кроме, возможно, петель нулевого веса, которые мы будем игнорировать). При изложении мы будем неявно предполагать, что параллельные ребра отсутствуют, и считать, что обозначение v->w однозначно указывает на ребро из v в w, хотя наш код справляется с ними без проблем.

Дерево кратчайших путей

Мы будем рассматривать *задачу кратчайших путей из одного источника*, где задана исходная вершина s. Результат вычислений представляет собой дерево, которое называется *деревом кратчайших путей* (ДКП) и содержит кратчайшие пути из s до любой вершины, достижимой из s.

Определение. Для заданного орграфа с взвешенными ребрами и заданной вершины s *деревом кратчайших путей* для источника s является подграф, содержащий вершину s и все достижимые из нее вершины, которые образуют ориентированное дерево с корнем в s, такое, что каждый путь в этом дереве является кратчайшим путем в орграфе (рис. 4.4.2 и 4.4.3).

Такое дерево всегда существует, но в общем случае могут присутствовать несколько путей одинаковой длины, соединяющих s с какой-то вершиной. В такой ситуации можно удалить последнее ребро в одном из этих путей и продолжать так, пока не останется только один путь, соединяющий источник с каждой вершиной (дерево с корнем). Когда построено дерево кратчайших путей, клиентам можно выдавать кратчайшие пути из s в любую вершину графа, используя представление родительскими ссылками — точно так же, как для путей в графах в разделе 4.1.

представление массивом родительских ребер

источник

0	null	6->0	6->0	6->0	6->0	6->1	6->0	6->0
1	5->1	null	5->1	5->1	5->1	5->1	5->1	5->1
2	0->2	6->2	null	6->2	6->2	6->2	6->2	6->2
3	7->3	1->3	1->3	null	7->3	1->3	7->3	7->3
4	0->4	6->4	5->4	6->4	null	5->4	6->4	5->4
5	4->5	7->5	7->5	7->5	4->5	null	7->5	7->5
6	3->6	3->6	3->6	3->6	3->6	3->6	null	3->6
7	2->7	2->7	2->7	2->7	4->7	5->7	2->7	null

Рис. 4.4.3. *Деревья кратчайших путей*

Типы данных орграфа с взвешенными ребрами

Наш тип данных для ориентированных ребер проще, чем для неориентированных, т.к. проход по ориентированным ребрам возможен только в одном направлении. Вместо методов either() и other(), которые были в классе Edge, здесь имеются методы from() и to() (рис. 4.4.4).

public class **DirectedEdge**		
	DirectedEdge(int v, int w, double weight)	
double	weight()	*вес данного ребра*
int	from()	*начальная вершина данного ребра*
int	to()	*конечная вершина данного ребра*
String	toString()	*строковое представление*

Рис. 4.4.4. *API взвешенного ориентированного ребра*

Аналогично переходу от класса Graph в разделе 4.1 к классу EdgeWeightedGraph из раздела 4.3, мы добавим метод edges() и будем использовать вместо целых чисел объекты DirectedEdge — см. рис. 4.4.5.

public class **EdgeWeightedDigraph**		
	EdgeWeightedDigraph(int V)	*пустой орграф с V вершинами*
	EdgeWeightedDigraph(In in)	*создание из потока* in
int	V()	*количество вершин*
int	E()	*количество ребер*
void	addEdge(DirectedEdge e)	*добавление в орграф ребра* e
Iterable<DirectedEdge>	adj(int v)	*ребра, направленные из* v
Iterable<DirectedEdge>	edges()	*все ребра данного орграфа*
String	toString()	*строковое представление*

Рис. 4.4.5. *API орграфа с взвешенными ребрами*

Реализации этих API приведены в листингах 4.4.1 и 4.4.2. Они представляют собой естественные расширения реализаций из разделов 4.2 и 4.3. Вместо списков смежности, содержащих целые числа, из класса Digraph в классе EdgeWeightedDigraph задействованы списки смежности, состоящие из объектов DirectedEdge. Как и в случае перехода от класса Graph в разделе 4.1 к классу Digraph из раздела 4.2, переход от класса EdgeWeightedGraph в разделе 4.3 к классу EdgeWeightedDigraph в данном разделе упрощает код, т.к. каждое ребро в такой структуре представлено только один раз.

Листинг 4.4.1. Тип данных ориентированного взвешенного ребра

```
public class DirectedEdge
{
   private final int v;                        // начальная вершина ребра
   private final int w;                        // конечная вершина ребра
```

```
  private final double weight;              // вес ребра

  public DirectedEdge(int v, int w, double weight)
  {
     this.v = v;
     this.w = w;
     this.weight = weight;
  }

  public double weight()
  {  return weight;  }

  public int from()
  {  return v;  }

  public int to()
  {  return w;  }

  public String toString()
  {  return String.format("%d->%d %.2f", v, w, weight);  }
}
```

Эта реализация DirectedEdge проще, чем реализация Edge для неориентированных взвешенных ребер Edge из раздела 4.3 (листинг 4.3.1), т.к. вершины ребра различаются. Для обращения к вершинам ребра e типа DirectedEdge в клиентах будут применяться идиомы int v = e.to(), w = e.from();.

Листинг 4.4.2. Тип данных орграфа с взвешенными ребрами

```
public class  EdgeWeightedDigraph
{
  private final int V;                  // количество вершин
  private int E;                        // количество ребер
  private Bag<DirectedEdge>[] adj;      // списки смежности

  public EdgeWeightedDigraph(int V)
  {
     this.V = V;
     this.E = 0;
     adj = (Bag<DirectedEdge>[]) new Bag[V];
     for (int v = 0; v < V; v++)
        adj[v] = new Bag<DirectedEdge>();
  }

  public EdgeWeightedDigraph(In in)
  // См. упражнение 4.4.2.

  public int V() {  return V;  }
  public int E() {  return E;  }

  public void addEdge(DirectedEdge e)
  {
     adj[e.from()].add(e);
     E++;
  }

  public Iterable<Edge> adj(int v)
  {  return adj[v];  }
```

```
  public Iterable<DirectedEdge> edges()
  {
    Bag<DirectedEdge> bag = new Bag<DirectedEdge>();
    for (int v = 0; v < V; v++)
      for (DirectedEdge e : adj[v])
        bag.add(e);
    return bag;
  }
}
```

Эта реализация EdgeWeightedDigraph представляет собой сплав типов EdgeWeightedGraph и Digraph, в котором используется индексированный вершинами массив контейнеров с объектами DirectedEdge. Как и в типе Digraph, каждое ребро представлено только один раз: если ребро направлено из v в w, то оно находится в списке смежности вершины v. Возможны петли и параллельные ребра. Реализация метода toString() оставлена на самостоятельную проработку в упражнении 4.4.2.

На рис. 4.4.6 изображена структура данных, построенная конструктором EdgeWeightedDigraph из ребер в левой части этого рисунка в указанном порядке. Как обычно, списки смежности представлены типом Bag и изображаются в виде связных списков (стандартная реализация). Как в случае невзвешенных орграфов из раздела 4.2, эта структура данных содержит только одно представление каждого ребра.

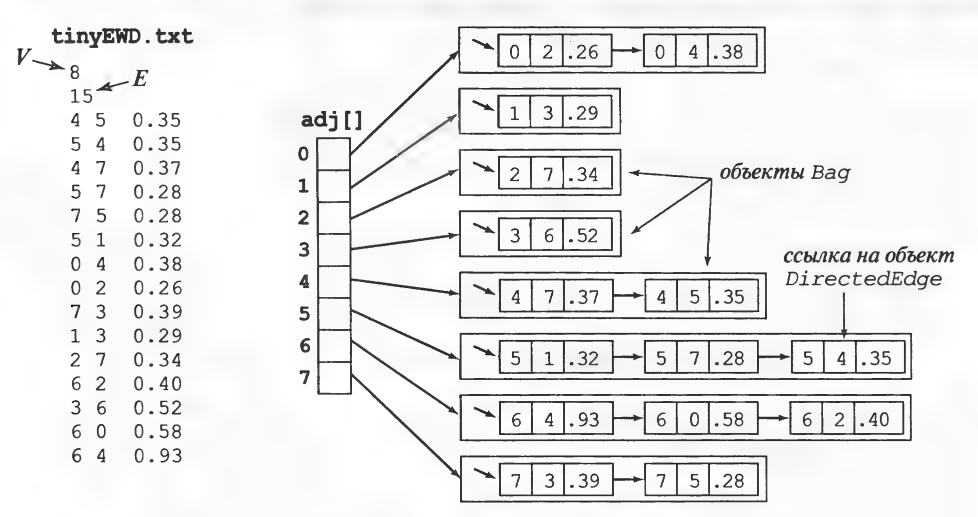


Рис. 4.4.6. Представление орграфа с взвешенными ребрами

API для кратчайших путей

Для нахождения кратчайших путей мы будем использовать ту же парадигму построения, что и для API DepthFirstPaths и BreadthFirstPaths из раздела 4.1. Чтобы предоставлять клиентам кратчайшие пути и их длины, наши алгоритмы будут реализовывать API, показанный на рис. 4.4.7.

public class SP

	`SP(EdgeWeightedDigraph G, int s)`	*конструктор*
`double`	`distTo(int v)`	*расстояние от* `s` *до* `v`*;* *∞, если путь не существует*
`boolean`	`hasPathTo(int v)`	*существует ли путь от* `s` *до* `v`*?*
`Iterable<DirectedEdge>`	`pathTo(int v)`	*путь от* `s` *до* `v`*;* `null`*, если путь не существует*

Рис. 4.4.7. API для реализаций поиска кратчайших путей

Конструктор строит дерево кратчайших путей и вычисляет длины этих кратчайших путей. Методы клиентских запросов используют эти структуры данных для предоставления клиентам длин и путей (с возможностью последовательного перебора ребер).

Клиент тестирования

Пример клиента приведен в листинге 4.4.3. Он принимает из командной строки в качестве аргументов входной поток и исходную вершину, читает из входного потока орграф с взвешенными ребрами, вычисляет ДКП для этого орграфа и источника и выводит кратчайший путь из источника до каждой другой вершины графа. Мы будем считать, что все наши реализации поиска кратчайших путей содержат этот клиент тестирования. В наших примерах будет использоваться файл `tinyEWD.txt` (см. рис. 4.4.6), определяющий ребра и веса небольшого демонстрационного орграфа, который мы будем использовать для выполнения подробных трассировок работы алгоритмов при поиске кратчайших путей. В нем используется тот же формат файла, что и для алгоритмов нахождения минимального остовного дерева (МОД): количество вершин V, за ним количество ребер E, а затем E строк, каждая из которых содержит два индекса вершин и вес. На сайте книги содержатся и файлы, определяющие несколько более крупных орграфов с взвешенными ребрами, в том числе и файл `mediumEWD.txt`, который определяет граф с 250 вершинами, изображенный на рис. 4.4.2. На рисунке каждый отрезок представляет ребра в обоих направлениях, так что этот файл содержит вдвое больше ребер, чем соответствующий файл `mediumEWG.txt`, на примере которого мы работали с МОД. На рисунках ДКП каждая линия представляет ориентированное ребро, направленное от источника.

Листинг 4.4.3. Клиент тестирования кратчайших путей

```
public static void main(String[] args)
{
   EdgeWeightedDigraph G;
   G = new EdgeWeightedDigraph(new In(args[0]));
   int s = Integer.parseInt(args[1]);
   SP sp = new SP(G, s);

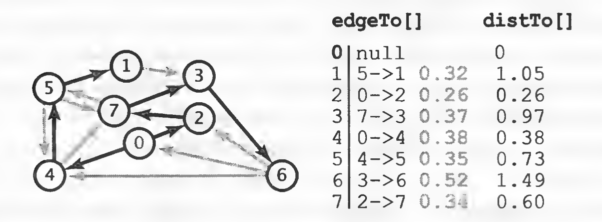
   for (int t = 0; t < G.V(); t++)
   {
      StdOut.print(s + " to " + t);
      StdOut.printf(" (%4.2f): ", sp.distTo(t));
      if (sp.hasPathTo(t))
         for (DirectedEdge e : sp.pathTo(t))
            StdOut.print(e + "   ");
      StdOut.println();
   }
}
```

```
% java SP tinyEWD.txt 0
0 to 0 (0.00):
0 to 1 (1.05): 0->4 0.38  4->5 0.35  5->1 0.32
0 to 2 (0.26): 0->2 0.26
0 to 3 (0.99): 0->2 0.26  2->7 0.34  7->3 0.39
0 to 4 (0.38): 0->4 0.38
0 to 5 (0.73): 0->4 0.38  4->5 0.35
0 to 6 (1.51): 0->2 0.26  2->7 0.34  7->3 0.39  3->6 0.52
0 to 7 (0.60): 0->2 0.26  2->7 0.34
```

Структуры данных для кратчайших путей

Структуры данных, необходимые для представления кратчайших путей, очевидны (рис. 4.4.8).

- *Ребра дерева кратчайших путей*: как в случаях поиска в глубину, поиска в ширину и алгоритма Прима, мы будем пользоваться представлением родительскими ребрами в виде индексированного вершинами массива edgeTo[] объектов DirectedEdge, где элемент edgeTo[v] — ребро, соединяющее вершину v с ее родителем в дереве (последнее ребро в кратчайшем пути из s в v).
- *Расстояние до источника*: мы будем пользоваться индексированным вершинами массивом distTo[] — таким, что элемент distTo[v] содержит длину кратчайшего известного пути из s в v.



	edgeTo[]		distTo[]
0	null		0
1	5->1	0.32	1.05
2	0->2	0.26	0.26
3	7->3	0.37	0.97
4	0->4	0.38	0.38
5	4->5	0.35	0.73
6	3->6	0.52	1.49
7	2->7	0.34	0.60

Рис. 4.4.8. Структуры данных кратчайшего пути

По соглашению edgeTo[s] содержит null, а distTo[s] равно 0. Кроме того, мы будем придерживаться соглашения, что расстояния до вершин, не достижимых из источника, равны Double.POSITIVE_INFINITY. Как обычно, мы разработаем типы данных, из которых конструктор будет строить эти структуры. Конечно, мы напишем и соответствующие методы экземпляров, которые позволят отвечать на клиентские запросы относительно кратчайших путей и их длин.

Релаксация ребра

Наши реализации поиска кратчайших путей основаны на простой операции, которая называется *релаксация*. Вначале нам известны только ребра графа и их веса, элемент distTo[] для источника равен 0, а во все остальные элементы занесено значение Double.POSITIVE_INFINITY. По мере работы алгоритма он собирает информацию о кратчайших путях, которые соединяют источник с каждой вершиной, встреченной в структурах данных edgeTo[] и distTo[]. При изменении этой информации при про-

смотре ребер возникают новые сведения о кратчайших путях. А именно, мы выполняем *релаксацию ребер* — для ребра v->w это означает проверку, что наилучший известный путь из s в w проходит от s до v, а потом через ребро v->w, и, если этот так, то соответственно изменяется содержимое структур данных. Это действие выполняет код из листинга 4.4.4.

Листинг 4.4.4. Релаксация ребра

```
private void relax(DirectedEdge e)
{
   int v = e.from(), w = e.to();
   if (distTo[w] > distTo[v] + e.weight())
   {
      distTo[w] = distTo[v] + e.weight();
      edgeTo[w] = e;
   }
}
```

Наилучшее известное расстояние от w до v равно сумме distTo[v] и e.weight(), и если это значение не меньше, чем distTo[w], мы считаем ребро ***непригодным*** и игнорируем его, а если оно ***меньше***, мы изменяем структуры данных. На рис. 4.4.9 показаны два возможных исхода операции релаксации ребра. Ребро v->w либо непригодно (пример слева), и тогда данные не меняются, либо формирует более короткий путь до w (как в примере справа). Во втором случае меняется содержимое элементов edgeTo[w] и distTo[w], и это может привести к непригодности других ребер и создать новые пригодные ребра. Термин *релаксация* (ослабление) возник из физической аналогии: как будто на путь, соединяющий две вершины, натянута эластичная лента, и релаксация ребра похожа на ослабление натяжения этой ленты по более короткому пути (если это возможно). Мы говорим, что ребро e можно ***успешно ослабить***, если вызов relax() изменяет значения distTo[e.to()] и edgeTo[e.to()].

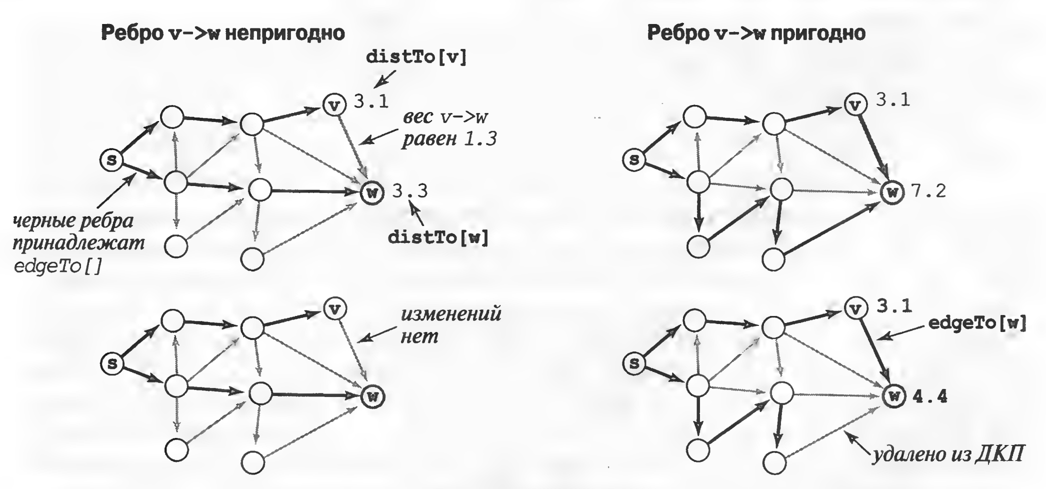


Рис. 4.4.9. Релаксация ребра (два случая)

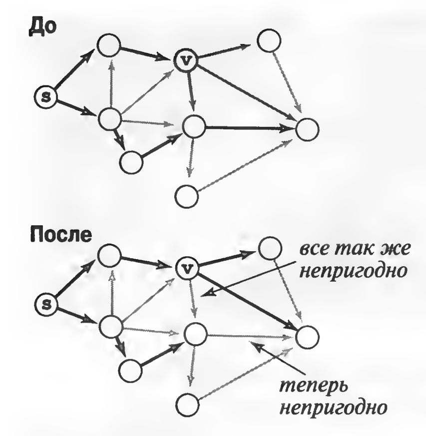


Рис. 4.4.10. Релаксация вершины

Релаксация вершины

На самом деле все наши реализации ослабляют *все* ребра, направленные из некоторой вершины — это видно из (перегруженной) реализации relax(), приведенной в листинге 4.4.5. Обратите внимание, что любое ребро из вершины с конечным значением distTo[] до вершины с бесконечным distTo[] является пригодным и добавляется в массив edgeTo[] при релаксации. В частности, первым добавляется в edgeTo[] какое-то ребро, направленное из источника. Наши алгоритмы аккуратно выбирают вершины, чтобы каждая релаксация ребра находила кратчайший (пока) путь до некоторой вершины, и постепенно подбираются к цели нахождения кратчайших путей для всех вершин (рис. 4.4.10).

Листинг 4.4.5. Релаксация вершины

```
private void relax(EdgeWeightedDigraph G, int v)
{
   for (DirectedEdge e : G.adj(v))
   {
      int w = e.to();
      if (distTo[w] > distTo[v] + e.weight())
      {
         distTo[w] = distTo[v] + e.weight();
         edgeTo[w] = e;
      }
   }
}
```

Методы клиентских запросов

Аналогично нашим реализациям для API поиска пути из раздела 4.1 (и упражнения 4.1.13), структуры данных edgeTo[] и distTo[] непосредственно поддерживают методы клиентских запросов pathTo(), hasPathTo() и distTo() (см. листинг 4.4.6). Этот код будет включен во все наши реализации поиска кратчайших путей. Как уже было сказано, значение distTo[v] имеет смысл только тогда, когда вершина v достижима из s, и мы будем придерживаться соглашения, что для вершин, не достижимых из s, метод distTo() возвращает бесконечность. Для этого мы вначале инициализируем все элементы distTo[] значениями Double.POSITIVE_INFINITY, кроме distTo[s] = 0. Наши реализации поиска кратчайших путей будут заносить в distTo[v] конечное значение для всех вершин v, которые достижимы из источника. Тогда мы сможем обойтись без массива marked[], который мы использовали для пометки достижимых вершин при поиске на графе, и в реализации hasPathTo(v) проверять distTo[v] на равенство Double.POSITIVE_INFINITY. В методе PathTo() мы будем придерживаться соглашения, что PathTo(v) возвращает null, если вершина v не достижима из источника, и путь без ребер, если v является источником. Для достижимых вершин будет выполняться проход вверх по дереву с заталкиванием встреченных ребер в стек — так же, как мы делали в реализациях DepthFirstPaths и BreadthFirstPaths. На рис. 4.4.11 показано обнаружение пути 0->2->7->3->6 для нашего примера.

Листинг 4.4.6. Методы клиентских запросов для кратчайших путей

```
public double distTo(int v)
{  return distTo[v];  }

public boolean hasPathTo(int v)
{  return distTo[v] < Double.POSITIVE_INFINITY;  }

public Iterable<DirectedEdge> pathTo(int v)
{
   if (!hasPathTo(v)) return null;
   Stack<DirectedEdge> path = new Stack<DirectedEdge>();
   for (DirectedEdge e = edgeTo[v]; e != null; e = edgeTo[e.from()])
      path.push(e);
   return path;
}
```

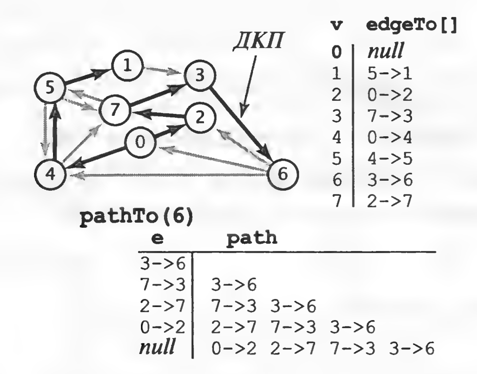


Рис. 4.4.11. Трассировка работы метода `pathTo()`

Теоретические основы разработки алгоритмов поиска кратчайших путей

Релаксация ребра — легкая для реализации фундаментальная операция, на которой основаны наши реализации поиска кратчайших путей. Но эта операция предоставляет и теоретическую основу для понимания алгоритмов, а при необходимости и для доказательства их корректности.

Условия оптимальности

Следующее утверждение доказывает эквивалентность *глобального* условия, что расстояния представляют собой длины кратчайших путей, и *локального* условия, что мы проверяем релаксацию ребра.

Утверждение P (условия оптимальности кратчайших путей). Пусть G — орграф с взвешенными ребрами, `s` — исходная вершина в G, а `distTo[]` — индексированный вершинами массив длин путей в G, таких, что для всех вершин `v`, достижимых из `s`, элемент `distTo[v]` содержит длину *некоторого* пути из `s` до `v`, причем `distTo[v]` равно бесконечности для всех `v`, не достижимых из `s`. Эти значения представляют собой длины *кратчайших* путей тогда и только тогда, когда они удовлетворяют условию `distTo[w] <= distTo[v] + e.weight()` для каждого ребра `e` на пути из `v` до `w` (т.е. ни одно ребро не является пригодным).

Доказательство. Пусть distTo[w] содержит длину кратчайшего пути из s до v. Если для некоторого ребра e на пути из v до w верно, что distTo[w] > distTo[v] + e.weight(), то ребро e образует путь из s до w (через v), длина которого меньше distTo[w]. Получено противоречие, которое означает, что условия оптимальности обязательны.

Чтобы доказать, что условия оптимальности достаточны, предположим, что вершина w достижима из s и что s = v_0->v_1->v_2...->v_k = w — кратчайший путь из s до w длиной OPT_{sw}. Обозначим ребро от v_{i-1} до v_i как e_i. В соответствии с условиями оптимальности мы имеем следующую последовательность неравенств:

$$
\begin{aligned}
distTo[w] = \; & distTo[v_k] && <= distTo[v_{k-1}] && + e_k.weight() \\
& distTo[v_{k-1}] && <= distTo[v_{k-2}] && + e_{k-1}.weight() \\
& \ldots \\
& distTo[v_2] && <= distTo[v_1] && + e_2.weight() \\
& distTo[v_1] && <= distTo[s] && + e_1.weight()
\end{aligned}
$$

Свернув эти неравенства и исключив distTo[s] = 0.0, получим

$$distTo[w] <= e_1.weight() + \ldots + e_k.weight() = OPT_{sw}.$$

Но distTo[w] — длина *некоторого* пути из s до w, поэтому она не может быть меньше длины *кратчайшего* пути. Итак, мы показали, что

$$OPT_{sw} <= distTo[w] <= OPT_{sw}$$

и должно выполняться равенство.

Проверка

Из утверждения P вытекает важное практическое следствие: его можно применять как средство проверки. Во время вычисления алгоритмом содержимого distTo[] можно в любой момент проверить, действительно ли он содержит длины кратчайших путей, выполнив один проход по ребрам графа и проверив, выполнены ли условия оптимальности. Алгоритмы поиска кратчайших путей могут быть сложными, и такая возможность проверить результат их работы может весьма пригодиться. На сайте книги мы выложили метод check(), предназначенный для этой цели. Этот метод проверяет также, что массив edgeTo[] определяет пути от источника и что он согласован с содержимым distTo[].

Обобщенный алгоритм

Из условий оптимальности непосредственно следует обобщенный алгоритм, который охватывает все алгоритмы нахождения кратчайших путей, которые мы рассмотрим. Пока мы ограничимся лишь неотрицательными весами.

Утверждение С (обобщенный алгоритм поиска кратчайших путей). Вначале заносим 0 в distTo[s] и бесконечность во все остальные элементы distTo[], а потом выполняем следующее действие.

Выполнить релаксацию любого ребра в G и повторять, пока есть пригодные ребра.

Для всех вершин w, достижимых из s, элемент distTo[w] после этого вычисления содержит длину кратчайшего пути из s до w (а элемент edgeTo[w] содержит последнее ребро этого пути).

Доказательство. Релаксация ребра v->w всегда заносит в distTo[w] длину некоторого пути из s (а в edgeTo[w] — последнее ребро этого пути). Для любой вершины w, достижимой из s, некоторое ребро из кратчайшего пути до w пригодно для релаксации, если элемент distTo[w] содержит бесконечность. Поэтому алгоритм продолжает работу, пока значение distTo[] для каждой вершины, достижимой из s, не станет длиной некоторого пути до этой вершины. На протяжении всей работы алгоритма для любой вершины v, для которой определен кратчайший путь, значение distTo[v] равно длине некоторого (простого) пути из s до v и строго монотонно убывает. Поэтому оно может уменьшиться лишь конечное количество раз (по разу для каждого простого пути из s до v). Когда не останется ни одного пригодного ребра, можно применить утверждение Р.

Основной причиной, по которой стоит рассматривать условия оптимальности и обобщенный алгоритм, является то, что обобщенный алгоритм *не задает порядок, в котором выполняется релаксация ребер*. Поэтому для доказательства, что любой алгоритм вычисляет кратчайшие пути, достаточно доказать, что он выполняет релаксации, пока есть пригодные для этого ребра.

Алгоритм Дейкстры

В разделе 4.3 был рассмотрен алгоритм Прима для нахождения МОД для неориентированного графа с взвешенными ребрами: на каждом шаге к единому растущему дереву добавляется новое ребро. *Алгоритм Дейкстры* (Dijkstra) вычисляет ДКП на основе аналогичной схемы. Вначале в dist[s] заносится 0, а во все остальные элементы distTo[] — положительная бесконечность, а затем мы *выполняем релаксацию и добавляем в дерево вершину извне дерева с наименьшим значением distTo[] и продолжаем так, пока все вершины не будут добавлены в дерево или пока ни одна вершина вне дерева не будет иметь конечное значение distTo[].*

Утверждение Т. Алгоритм Дейкстры решает задачу нахождения кратчайших путей из одного источника в орграфах с взвешенными ребрами, веса которых неотрицательны.

Доказательство. Если вершина v достижима из источника, то для каждого ребра v->w релаксация выполняется только один раз, причем после релаксации distTo[w] <= distTo[v] + e.weight(). Это неравенство выполняется до завершения алгоритма, т.к. distTo[w] может только убывать (любая релаксация может лишь уменьшить значение distTo[]), а значение distTo[v] не меняется (веса ребер неотрицательны, и на каждом шаге мы выбираем наименьшее значение distTo[], поэтому никакая последующая релаксация не может занести в любой элемент distTo[] значение, меньшее distTo[v]). Значит, после добавления в дерево всех вершин, достижимых из s, условие оптимальность кратчайших путей выполняется, и можно применить утверждение Р.

Структуры данных

Для реализации алгоритма Дейкстры мы добавим к структурам данных distTo[] и edgeTo[] индексную очередь с приоритетами pq, в которой будут храниться вершины-кандидаты на следующую релаксацию. Вспомните, что класс IndexMinPQ позволяет связывать индексы с ключами (приоритетами) и извлекать индекс, соответствующий наименьшему ключу. В данном случае мы всегда связываем вершину v со значением distTo[v] — и получаем непосредственную реализацию алгоритма Дейкстры, вытекающую из его формулировки. Более того, по индукции следует, что элементы edgeTo[], соответствующие достижимым вершинам, образуют дерево — дерево кратчайших путей.

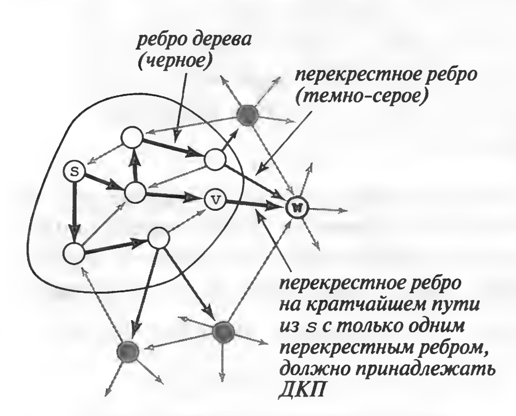


Рис. 4.4.12. Алгоритм Дейкстры для нахождения кратчайших путей

Альтернативный взгляд

Другой способ понять динамику алгоритма, который вытекает из доказательства, приведен на рис. 4.4.12: у нас имеется инвариант, что элементы distTo[] для вершин дерева представляют собой расстояния вдоль кратчайших путей, и для каждой вершины w из очереди с приоритетами значение distTo[w] равно весу кратчайшего пути из s до w, который использует только промежуточные вершины в дереве и заканчивается перекрестным ребром edgeTo[w]. Элемент distTo[] для вершины с *наименьшим* приоритетом — это вес кратчайшего пути, не меньший веса кратчайшего пути до любой вершины, для которой выполнена релаксация, и не больший веса кратчайшего пути до любой вершины, для которой релаксация еще не выполнена. И следующая релаксация должна быть выполнена для этой вершины. Для достижимых вершин релаксация выполняется в порядке весов их кратчайших путей от s.

На рис. 4.4.13 приведена трассировка для нашего маленького демонстрационного графа tinyEWD.txt. Для этого примера алгоритм строит ДКП следующим образом:

- Добавляет вершину 0 в дерево, а смежные с ней вершины 2 и 4 — в очередь с приоритетами.
- Удаляет вершину 2 из очереди с приоритетами, заносит ребро 0->2 в дерево, а затем добавляет вершину 7 в очередь с приоритетами.
- Удаляет вершину 4 из очереди с приоритетами, заносит ребро 0->4 в дерево, а затем добавляет вершину 5 в очередь с приоритетами. Ребро 4->7 непригодно.
- Удаляет вершину 7 из очереди с приоритетами, заносит ребро 2->7 в дерево, а затем добавляет вершину 3 в очередь с приоритетами. Ребро 7->5 непригодно.
- Удаляет вершину 5 из очереди с приоритетами, заносит ребро 4->5 в дерево, а затем добавляет вершину 1 в очередь с приоритетами. Ребро 5->7 непригодно.
- Удаляет вершину 3 из очереди с приоритетами, заносит ребро 7->3 в дерево, а затем добавляет вершину 6 в очередь с приоритетами.
- Удаляет вершину 1 из очереди с приоритетами и заносит ребро 5->1 в дерево. Ребро 1->3 непригодно.
- Удаляет вершину 6 из очереди с приоритетами и заносит ребро 3->6 в дерево.

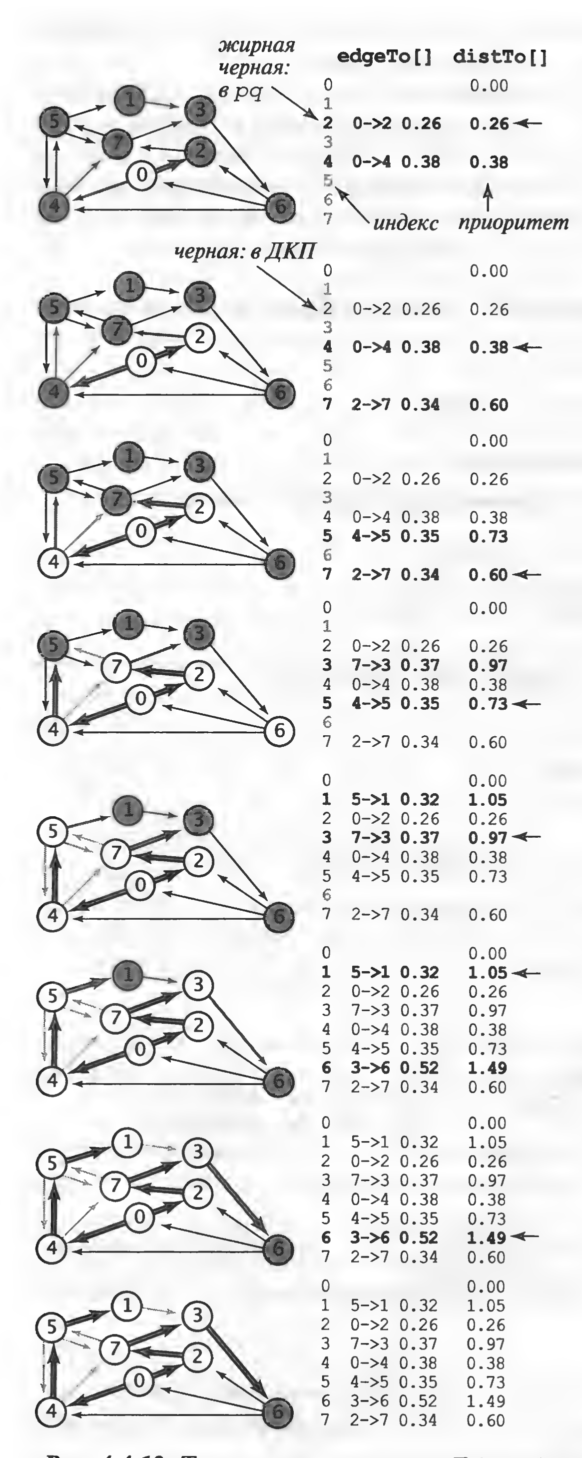


Рис. 4.4.13. Трассировка алгоритма Дейкстры

Вершины добавляются в ДКП в порядке увеличения их расстояния от источника — на рисунке они отмечены стрелками справа.

Реализация алгоритма Дейкстры DijkstraSP (см. листинг 4.4.7) представляет собой программную запись нашего краткого описания алгоритма и позволяет, после добавления одного оператора в метод relax(), обрабатывать два случая: либо вершина to() некоторого ребра еще не занесена в очередь с приоритетами, и в этом случае она заносится туда с помощью метода insert(), либо она уже находится в очереди с приоритетами, и ее приоритет понижается — это делает метод change().

Листинг 4.4.7. Алгоритм 4.9. Алгоритм Дейкстры для нахождения кратчайших путей

```
public class  DijkstraSP
{
   private DirectedEdge[] edgeTo;
   private double[] distTo;
   private IndexMinPQ<Double> pq;

   public DijkstraSP(EdgeWeightedDigraph G, int s)
   {
      edgeTo = new DirectedEdge[G.V()];
      distTo = new double[G.V()];
      pq = new IndexMinPQ<Double>(G.V());

      for (int v = 0; v < G.V(); v++)
         distTo[v] = Double.POSITIVE_INFINITY;
      distTo[s] = 0.0;

      pq.insert(s, 0.0);
      while (!pq.isEmpty())
         relax(G, pq.delMin())
   }

   private void relax(EdgeWeightedDigraph G, int v)
   {
      for(DirectedEdge e : G.adj(v))
      {
         int w = e.to();
         if (distTo[w] > distTo[v] + e.weight())
         {
            distTo[w] = distTo[v] + e.weight();
            edgeTo[w] = e;
            if (pq.contains(w)) pq.change(w, distTo[w]);
            else                pq.insert(w, distTo[w]);
         }
      }
   }

   public double distTo(int v)           // Стандартные методы клиентских запросов
   public boolean hasPathTo(int v)       //   для реализаций ДКП
   public Iterable<Edge> pathTo(int v)   //   (см. листинг 4.4.6)
}
```

Эта реализация алгоритма Дейкстры выращивает ДКП, добавляя к нему по одному ребру и всегда выбирая для этого ребро из вершины дерева в вершину вне дерева, которая наиболее близка к s.

Утверждение Т (продолжение). Для вычисления ДКП с заданным корнем в орграфе с E взвешенными ребрами и V вершинами алгоритму Дейкстры требуется дополнительный объем памяти, пропорциональный $E \log V$ (в худшем случае).

Доказательство. Совпадает с доказательством алгоритма Прима (см. утверждение О).

Как уже было сказано, для понимания алгоритма Дейкстры его можно сравнить с алгоритмом Прима из раздела 4.3 (алгоритм 4.3.6). Оба эти алгоритма строят дерево с корнем, добавляя к нему по одному ребру. Алгоритм Прима добавляет к дереву очередную вершину извне этого дерева, которая ближе всего к *дереву*, тогда как алгоритм Дейкстры добавляет к дереву очередную вершину извне этого дерева, которая ближе всего к *источнику*. Массив `marked[]` не нужен, т.к. условие `!marked[w]` эквивалентно условию бесконечности значения `distTo[w]`. Другими словами, перейдя к неориентированным графам и ребрам и отбросив ссылки на `distTo[v]` в коде метода `relax()` из алгоритма 4.9, мы получим реализацию алгоритма 4.7 — "энергичный" вариант алгоритма Прима! Кроме того, совсем не трудно разработать "ленивый" вариант алгоритма Дейкстры, соответствующий коду `LazyPrimMST` (листинг 4.3.5).

Варианты

Наша реализация алгоритма Дейкстры (с соответствующими модификациями) годится для решения и других вариантов задачи — например, следующего.

Поиск кратчайших путей из одного источника в неориентированных графах. Для заданного *неориентированного* графа с взвешенными ребрами и исходной вершины `s` необходимо отвечать на запросы вида *Существует ли путь из* `s` *до указанной вершины* `v`*?* Если да, то нужно найти *кратчайший* такой путь (с минимальным общим весом).

Для решения этой задачи достаточно рассматривать неориентированный граф как орграф. То есть для заданного неориентированного графа можно построить орграф с взвешенными ребрами и теми же вершинами, но каждому ребру исходного графа соответствуют два ориентированных ребра орграфа (по одному в каждом направлении). Между путями в орграфе и в неориентированном графе существует взаимно однозначное соответствие, и стоимость путей одна и та же, т.е. эти задачи поиска кратчайших путей эквивалентны.

Кратчайшие пути из источника в сток. Для заданного орграфа с взвешенными ребрами, исходной вершины `s` и целевой вершины `t` необходимо найти кратчайший путь из `s` до `t`.

Для решения этой задачи используется алгоритм Дейкстры, но его работа прекращается сразу после извлечения вершины `t` из очереди с приоритетами.

Кратчайшие пути для всех пар вершин. Для заданного орграфа с взвешенными ребрами необходимо отвечать на запросы вида *Существует ли путь из источника* `s` *в целевую вершину* `t`*?* Если да, то нужно найти *кратчайший* такой путь (с минимальным весом).

На удивление компактная реализация в листинге 4.4.8 решает задачу поиска кратчайших путей для всех пар вершин, требуя объем памяти и время, пропорциональные $EV \log V$. Она строит массив объектов `DijkstraSP` — для каждой вершины в качестве источника. Для ответов на клиентские запросы используется источник для доступа к

соответствующему объекту кратчайших путей из одного источника, а затем в качестве аргумента запроса передается целевая вершина.

Листинг 4.4.8. Кратчайшие пути для всех пар вершин

```
public class  DijkstraAllPairsSP
{
  private DijkstraSP[] all;

  DijkstraAllPairsSP(EdgeWeightedDigraph G)
  {
    all = new DijkstraSP[G.V()]
    for (int v = 0; v < G.V(); v++)
      all[v] = new DijkstraSP(G, v);
  }

  Iterable<Edge> path(int s, int t)
  {  return all[s].pathTo(t);  }

  double dist(int s, int t)
  {  return all[s].distTo(t);  }
}
```

> ***Кратчайшие пути в евклидовых графах.*** Требуется решать задачи поиска кратчайших путей с одним источником, с одним стоком и для всех пар вершин в графах, где вершины представляют собой точки на плоскости, а веса ребер пропорциональны геометрическим расстояниям между вершинами.

В данном случае простая модификация может существенно ускорить работу алгоритма Дейкстры (см. упражнение 4.4.27).

На рис. 4.4.14 показано разрастание ДКП, вычисляемого алгоритмом Дейкстры для евклидова графа, который определен в файле `mediumEWD.txt`, для нескольких различных источников. Вспомните, что отрезки в этом графе представляют ориентированные ребра в обоих направлениях. Приведенные диаграммы демонстрируют красивый динамический процесс.

Теперь мы рассмотрим алгоритмы поиска кратчайших путей для ациклических графов с взвешенными ребрами, которые отрабатывают за линейное время (быстрее алгоритма Дейкстры), а потом для орграфов с ребрами, которые могут иметь отрицательные веса — для них алгоритм Дейкстры просто неприменим.

Ациклические орграфы с взвешенными ребрами

Во многих естественных приложениях орграфы с взвешенными ребрами точно не могут иметь ориентированных циклов. Для краткости ациклические орграфы с взвешенными ребрами мы будем называть *взвешенными ориентированными ациклическими графами* (рис. 4.4.15). И сейчас мы рассмотрим алгоритм поиска кратчайших путей, который проще и быстрее алгоритма Дейкстры при работе с взвешенными ориентированными ациклическими графами. Точнее, этот алгоритм:

- решает задачу для одного источника за линейное время;
- справляется с ребрами с отрицательными весами;
- решает родственные задачи — например, поиск *самых длинных* путей.

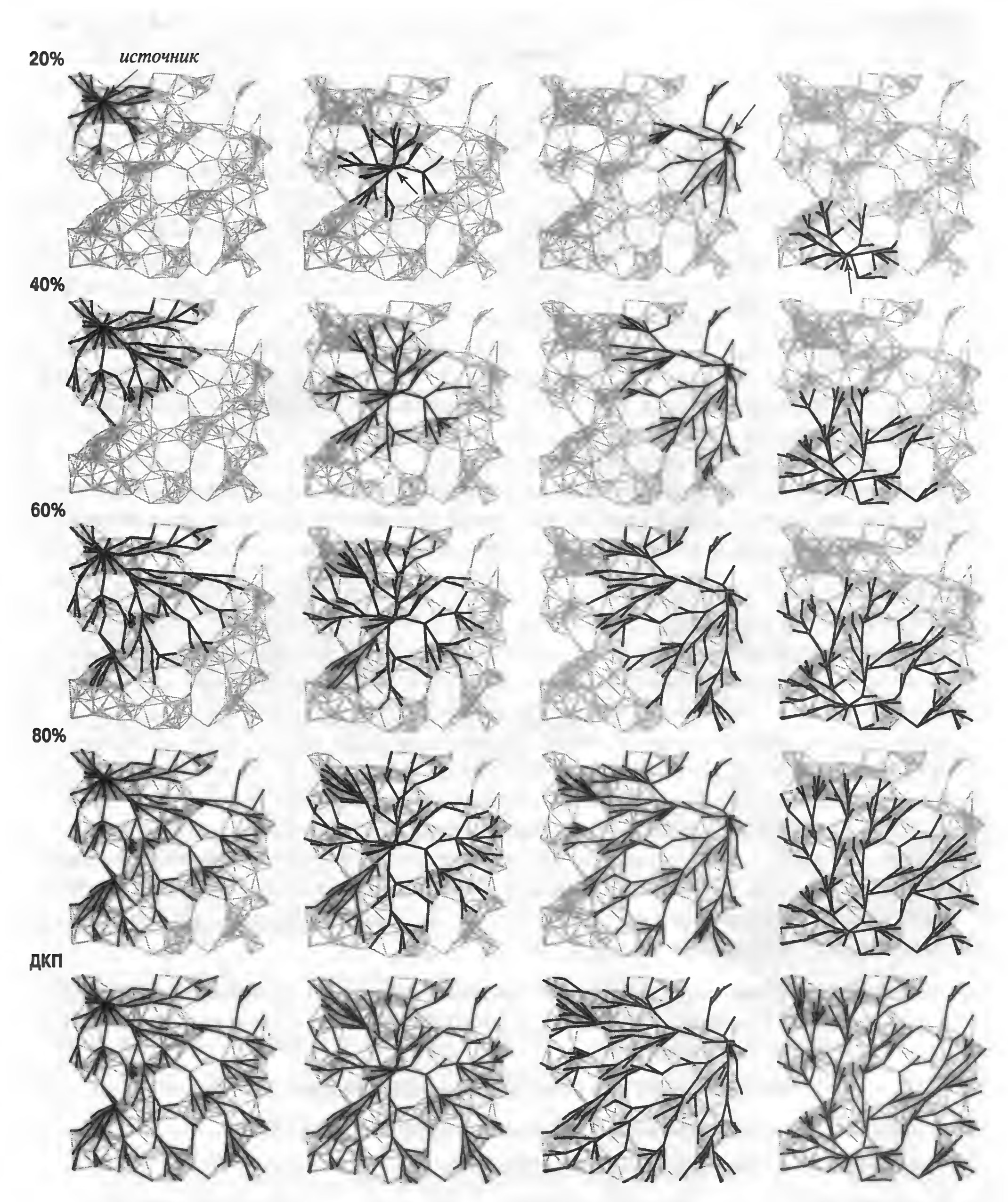


Рис. 4.4.14. Алгоритм Дейкстры (250 вершин, различные источники)

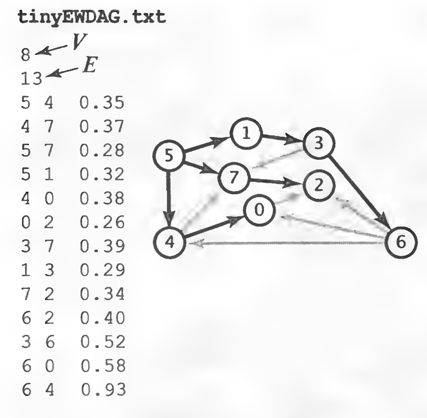


Рис. 4.4.15. Ациклический орграф с взвешенными ребрами и его ДКП

Эти алгоритмы представляют собой очевидные расширения алгоритма для топологической сортировки в ориентированных ациклических графах, который был рассмотрен в разделе 4.2.

А именно, релаксация вершин в сочетании с топологической сортировкой дает решение задачи поиска кратчайших путей из одного источника для взвешенных ориентированных ациклических графов. Вначале в distTo[s] заносится 0, а во все другие элементы distTo[] бесконечность, а потом поочередно выполняется релаксация вершин, причем вершины выбираются *в топологическом порядке*. Рассуждение, аналогичное (но еще проще) рассуждению для алгоритма Дейкстры, подтверждает эффективность этого способа.

Утверждение У. Выполнение релаксации вершин в топологическом порядке позволяет решить задачу поиска кратчайших путей из одного источника для взвешенных ориентированных ациклических графов за время, пропорциональное $E + V$.

Доказательство. Релаксация каждого ребра v->w выполняется только один раз, при релаксации вершины v, и при этом distTo[w] <= distTo[v] + e.weight(). Это неравенство выполняется до завершения работы алгоритма, т.к. значение distTo[v] не меняется (в силу топологической упорядоченности ни одно ребро, направленное в v, уже не будет обрабатываться после релаксации v), а величина distTo[w] может только убывать (любая релаксация может лишь уменьшить значение distTo[]). Поэтому после того как все вершины, достижимые из s, добавлены в дерево, будут выполняться условия оптимальности кратчайших путей, и можно применить утверждение С. Вывод граничного значения времени: утверждение Ж (раздел 4.2) гласит, что для выполнения топологической сортировки необходимо время, пропорциональное $E + V$, и второй проход релаксации завершает работу, выполняя релаксацию один раз для каждого ребра — на что снова нужно время, пропорциональное $E + V$.

На рис. 4.4.16 приведена трассировка для демонстрационного ациклического орграфа с взвешенными ребрами tinyEWDAG.txt. Алгоритм строит дерево кратчайших путей из вершины 5 следующим образом.

- Выполняет поиск в глубину для вычисления топологического порядка 5 1 3 6 4 7 0 2.
- Добавляет в дерево вершину 5 и все ребра, направленные из нее.
- Добавляет в дерево вершину 1 и ребро 1->3.
- Добавляет в дерево вершину 3 и ребро 3->6, но не 3->7, которое непригодно.
- Добавляет в дерево вершину 6 и ребра 6->2 и 6->0, но не 6->4, которое непригодно.
- Добавляет в дерево вершину 4 и ребро 4->0, но не 4->7, которое непригодно. Ребро 6->0 становится непригодным.
- Добавляет в дерево вершину 7 и ребро 7->2. Ребро 6->2 становится непригодным.

- Добавляет в дерево вершину 0, но не инцидентное ей ребро 0->2, которое непригодно.
- Добавляет в дерево вершину 2.

Добавление в дерево вершины 2 на рисунке не показано; у последней вершины в топологическом порядке нет направленных из нее ребер.

Реализация, приведенная в алгоритме 4.10 (листинг 4.4.9), представляет собой очевидное выражение в коде только что описанного процесса. В ней предполагается, что класс `Topological` содержит перегруженные методы для топологической сортировки и использует API `EdgeWeightedDigraph` и `DirectedEdge` из данного раздела (см. упражнение 4.4.12). Обратите внимание, что в этой реализации логический массив `marked[]` не нужен: поскольку обрабатываются вершины ациклического графа в топологическом порядке, алгоритм никогда не попадает в вершину, для которой уже выполнена релаксация. Вряд ли можно предложить что-то более эффективное, чем алгоритм 4.10: после топологической сортировки конструктор просматривает граф и выполняет релаксацию каждого ребра в точности один раз. Это рекомендуемый метод для нахождения кратчайших путей в графах с взвешенными ребрами, про которые точно известно, что они не содержат циклов.

Листинг 4.4.9. Алгоритм 4.10. Кратчайшие пути в ориентированном ациклическом графе с взвешенными ребрами

```
public class AcyclicSP
{
  private DirectedEdge[] edgeTo;
  private double[] distTo;

  public AcyclicSP(EdgeWeightedDigraph G, int s)
  {
     edgeTo = new DirectedEdge[G.V()];
     distTo = new double[G.V()];

     for (int v = 0; v < G.V(); v++)
        distTo[v] = Double.POSITIVE_INFINITY;
     distTo[s] = 0.0;

     Topological top = new Topological(G);

     for (int v : top.order())
        relax(G, v);
  }

  private void relax(EdgeWeightedDigraph G, int v)
  // См. листинг 4.4.5.
```

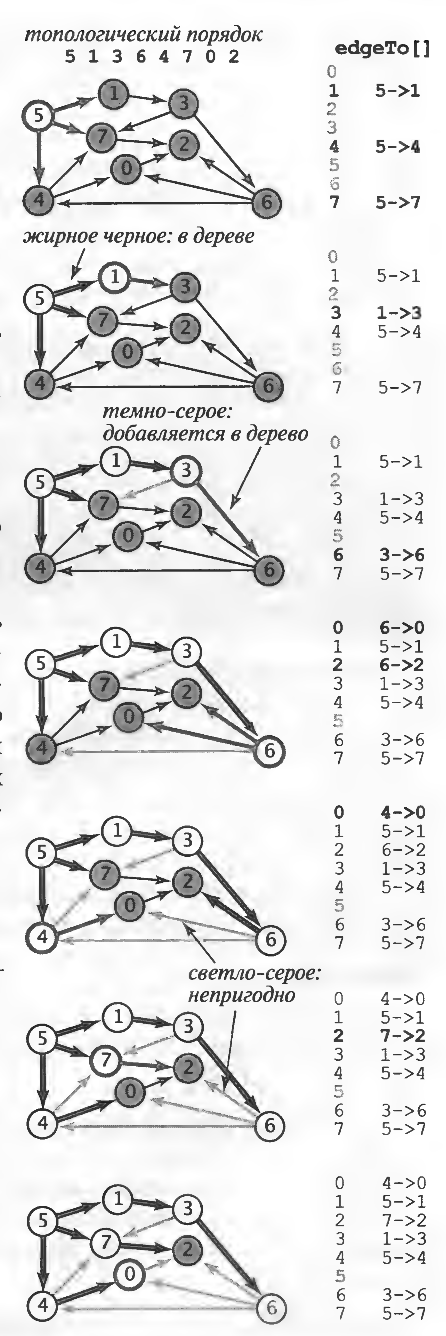


Рис. 4.4.16. Трассировка поиска кратчайших путей в ориентированном ациклическом графе с взвешенными ребрами

```
  public double distTo(int v)         // Стандартные методы клиентских запросов
  public boolean hasPathTo(int v)     //   для реализаций ДКП
  public Iterable<Edge> pathTo(int v) //   (см. листинг 4.4.6)
}
```

Этот алгоритм поиска кратчайших путей для взвешенных ориентированных ациклических графов применяет топологическую сортировку (алгоритм 4.5, адаптированный для использования типов EdgeWeightedDigraph и DirectedEdge), чтобы выполнять релаксацию вершин в топологическом порядке — и это все, что требуется для вычисления кратчайших путей.

```
% java AcyclicSP tinyEWDAG.txt 5
5 to 0 (0.73): 5->4 0.35  4->0 0.38
5 to 1 (0.32): 5->1 0.32
5 to 2 (0.62): 5->7 0.28  7->2 0.34
5 to 3 (0.62): 5->1 0.32  1->3 0.29
5 to 4 (0.35): 5->4 0.35
5 to 5 (0.00):
5 to 6 (1.13): 5->1 0.32  1->3 0.29  3->6 0.52
5 to 7 (0.28): 5->7 0.28
```

Утверждение У важно тем, что оно предоставляет конкретный пример, где отсутствие циклов существенно упрощает задачу. Для кратчайших путей методы, основанные на топологической упорядоченности, работают быстрее алгоритма Дейкстры на время, необходимое для выполнения операций в очереди с приоритетами. Более того, доказательство утверждения У не требует обязательной неотрицательности весов ребер, и мы можем удалить это ограничение для взвешенных ориентированных ациклических графов. Сейчас мы рассмотрим следствия из этой возможности отрицательных весов ребер, т.е. использование модели кратчайших путей для решения двух других задач, одна из которых на первый взгляд весьма далека от обработки графов.

Самые длинные пути

Рассмотрим задачу поиска *самого длинного* пути во взвешенном ориентированном ациклическом графе, ребра которого могут иметь как положительный, так и отрицательный вес.

Самые длинные пути из одного источника во взвешенных ориентированных ациклических графах. Пусть задан взвешенный ориентированный ациклический граф (с возможными отрицательными весами ребер) и исходная вершина s. Необходимо отвечать на запросы вида *Существует ли ориентированный путь из* s *до указанной вершины v?* Если да, нужно найти *самый длинный* такой путь (общий вес которого *максимален*).

Рассмотренный выше алгоритм легко справляется с этой задачей.

Утверждение Ф. Задачу поиска самых длинных путей во взвешенных ориентированных ациклических графах можно решить за время, пропорциональное $E + V$.

Доказательство. Создадим копию заданного взвешенного ориентированного ациклического графа, которая идентична исходному, но все веса имеют обратные знаки.

Тогда *кратчайший* путь в этой копии является *самым длинным* путем в исходном графе. Для преобразования найденного кратчайшего пути в решение задачи поиска самого длинного пути нужно изменить в решении знаки весов всех ребер. Время выполнения непосредственно следует из утверждения У.

На основе такого преобразования несложно разработать класс AcyclicLP, который находит самые длинные пути во взвешенных ориентированных ациклических графах. Но еще проще скопировать класс AcyclicSP, заменить в нем первоначальные значения distTo[] на Double.NEGATIVE_INFINITY и изменить знак неравенства в методе relax(). В любом случае получается эффективное решение задачи нахождения самых длинных путей во взвешенных ориентированных ациклических графах. Сравните этот результат с тем, что наилучший известный алгоритм для нахождения самых длинных простых путей во взвешенных орграфах общего вида (где веса ребер могут быть отрицательными) требует в худшем случае *экспоненциального* времени (см. главу 6)! Наличие циклов существенно усложняет задачу.

На рис. 4.4.17 приведена трассировка процесса поиска самых длинных путей в нашем демонстрационном взвешенном ориентированном ациклическом графе tinyEWDAG.txt, которую можно сравнить с трассировкой поиска кратчайших путей в том же самом графе на рис. 4.4.16. В этом примере алгоритм строит дерево самых длинных путей из вершины 5 следующим образом.

- Выполняет поиск в глубину для вычисления топологического порядка 5 1 3 6 4 7 0 2.
- Добавляет в дерево вершину 5 и все ребра, направленные из нее.
- Добавляет в дерево вершину 1 и ребро 1->3.
- Добавляет в дерево вершину 3 и ребра 3->6 и 3->7. Ребро 5->7 становится непригодным.
- Добавляет в дерево вершину 6 и ребра 6->2, 6->4 и 6->0.
- Добавляет в дерево вершину 4 и ребра 4->0 и 4->7. Ребра 6->0 и 3->7 становятся непригодными.

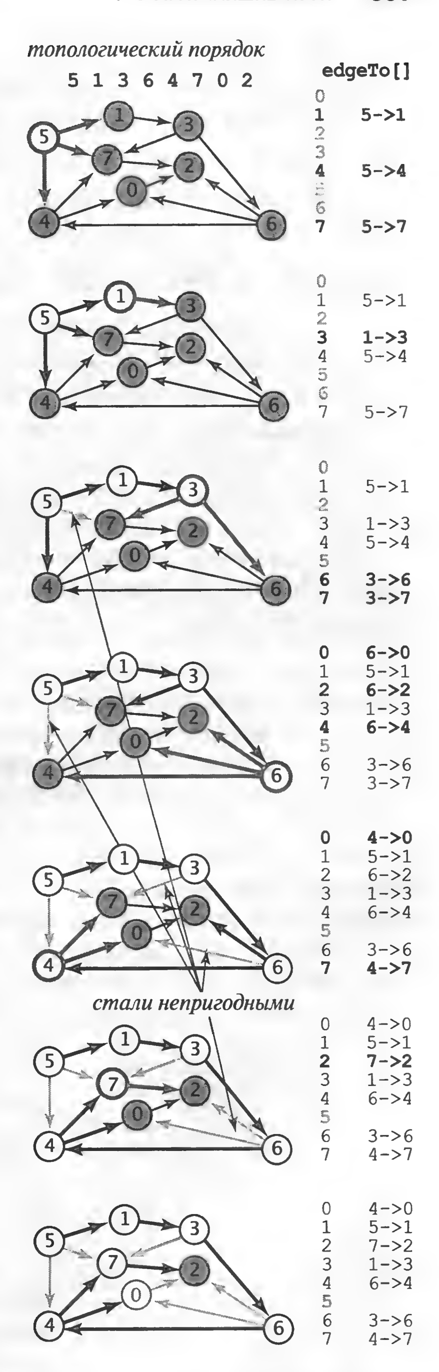


Рис. 4.4.17. Трассировка поиска самых длинных путей в ациклической сети

- Добавляет в дерево вершину 7 и ребро 7->2. Ребро 6->2 становится непригодным.
- Добавляет в дерево вершину 0, но без инцидентного ей ребра 0->2, которое непригодно.
- Добавляет в дерево вершину 2 (не показано на рисунке).

Алгоритм поиска самых длинных путей обрабатывает вершины в том же порядке, что и алгоритм для кратчайших путей, но дает совершенно другой результат.

Планирование параллельных работ

В качестве демонстрационного приложения мы еще раз обратимся к классу задач *планирования*, который мы уже рассматривали в разделе 4.2. А именно, мы рассмотрим следующую задачу планирования (отличия от задачи из раздела 4.2 выделены курсивом).

> ***Планирование параллельных работ с ограничениями предшествования.*** Имеется множество работ *заданной продолжительности*, которые нужно выполнить, и ограничения предшествования, которые указывают, что некоторые работы должны быть завершены, прежде чем можно начать некоторые другие работы. Как можно запланировать выполнение этих работ *на одинаковых процессорах* (*в любом нужном количестве*), чтобы все они завершились *за минимальное время* и с учетом ограничений?

В модели из раздела 4.2 неявно предполагается наличие одного процессора: работы выполняются в топологическом порядке, и общее время, необходимое для выполнения, равно общей продолжительности всех работ. Но сейчас мы предполагаем, что имеется достаточное количество процессоров для выполнения скольких угодно работ, и все ограничения относятся только к предшествованию. В принципе могут быть задействованы тысячи или даже миллионы работ, поэтому нам необходим эффективный алгоритм.

К счастью, существует алгоритм с *линейным* временем выполнения. Такой подход, называемый *методом критического пути*, демонстрирует задачу, эквивалентную задаче поиска самых длинных путей во взвешенном ориентированном ациклическом графе. Этот метод успешно применяется в бесчисленных производственных приложениях.

Мы постараемся запланировать выполнение каждой работы как можно раньше, предполагая, что любой доступный процессор может выполнить работу за время ее продолжительности. Например, рассмотрим пример задачи на рис. 4.4.18. Решение, приведенное на рис. 4.4.19, показывает, что минимально возможное время завершения для любого графика выполнения этой задачи равно 173.0: этот график удовлетворяет всем ограничениям, и ни один график не может завершиться раньше момента 173.0 из-за последовательности работ 0->9->6->8->2. Эта последовательность называется *критическим путем* для данной задачи. Любая последовательность работ, которые должны выполняться после работ, предшествующим им в последовательности, представляет нижнюю границу продолжительности всего графика. Если определить длину такой последовательности как наиболее раннее возможное время завершения (сумма длительностей работ), то самая длинная последовательность и будет критическим путем, т.к. любая задержка во времени начала любой работы сдвигает наилучшее возможное время завершения всего проекта.

работа	*продолжительность*	*должна быть выполнена до*
0	41.0	1 7 9
1	51.0	2
2	50.0	
3	36.0	
4	38.0	
5	45.0	
6	21.0	3 8
7	32.0	3 8
8	32.0	2
9	29.0	4 6

Рис. 4.4.18. *Задача планирования работ*

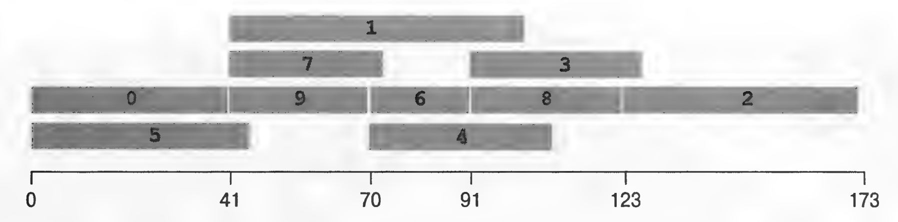


Рис. 4.4.19. Решение задачи параллельного планирования работ

Определение. Метод критического пути для параллельного планирования работ состоит в следующем. Создается взвешенный ориентированный ациклический граф с источником s, стоком t и двумя вершинами для каждой работы (*начальная* и *конечная* вершины). Для каждой работы добавляется ребро из ее начальной вершины в ее конечную вершину с весом, равным продолжительности этой работы. Для каждого ограничения предшествования v->w добавляется ребро нулевого веса из конечной вершины, соответствующей v, до начальной вершины, соответствующей w. Кроме того, добавляются ребра нулевого веса из источника до начальной вершины каждой работы и из конечной вершины каждой работы до стока. После этого каждая работа начинает выполняться в момент времени, определяемый длиной ее самого длинного пути из источника.

На рис. 4.4.20 изображено это соответствие для нашей демонстрационной задачи, а на рис. 4.4.21 приведены найденные самые длинные пути. Как было сказано, у каждой работы имеются три ребра — ребра нулевого веса от источника до начала и от конца до стока, и ребро от начала до конца — и еще по одному ребру для каждого ограничения предшествования. Класс CPM в листинге 4.4.10 представляет собой очевидную реализацию метода критического пути. Он преобразует любой экземпляр задачи планирования работ в экземпляр задачи поиска самых длинных путей во взвешенном ориентированном ациклическом графе, решает эту задачу с помощью класса AcyclicLP, а затем выводит моменты начала работ и запланированные моменты их завершения.

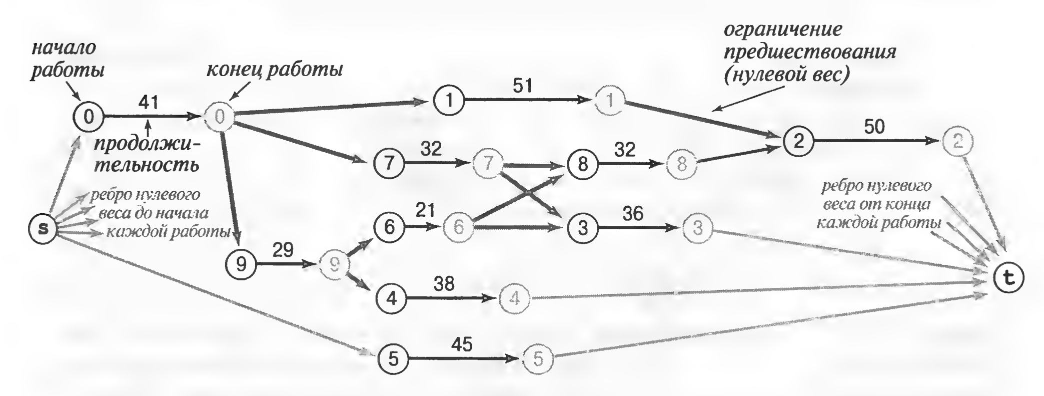


Рис. 4.4.20. Представление планирования работ с помощью взвешенного ориентированного ациклического графа

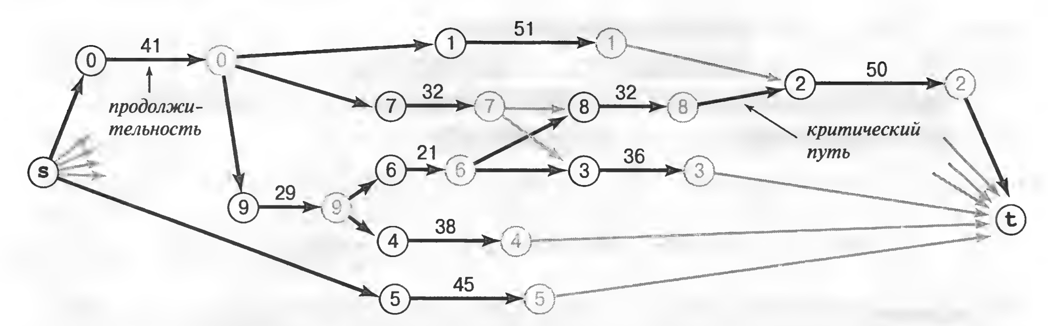


Рис. 4.4.21. Пример нахождения самого длинного пути для планирования работ

Листинг 4.4.10. Метод критического пути для планирования параллельных работ с ограничениями предшествования

```
public class  CPM
{
   public static void main(String[] args)
   {
      int N = StdIn.readInt(); StdIn.readLine();
      EdgeWeightedDigraph G;
      G = new EdgeWeightedDigraph(2*N+2);
      int s = 2*N, t = 2*N+1;
      for (int i = 0; i < N; i++)
      {
         String[] a = StdIn.readLine().split("\\s+");
         double duration = Double.parseDouble(a[0]);
         G.addEdge(new DirectedEdge(i, i+N, duration));
         G.addEdge(new DirectedEdge(s, i, 0.0));
         G.addEdge(new DirectedEdge(i+N, t, 0.0));
         for (int j = 1; j < a.length; j++)
         {
            int successor = Integer.parseInt(a[j]);
            G.addEdge(new DirectedEdge(i+N, successor, 0.0));
         }
      }
      AcyclicLP lp = new AcyclicLP(G, s);
      StdOut.println("Моменты начала:");
      for (int i = 0; i < N; i++)
         StdOut.printf("%4d: %5.1f\n", i, lp.distTo(i));
      StdOut.printf("Момент завершения: %5.1f\n", lp.distTo(t));
   }
}
```

Данная реализация метода критического пути сводит задачу планирования работ непосредственно к задаче поиска самых длинных путей во взвешенном ориентированном ациклическом графе. Согласно описанию метода критического пути, она строит орграф с взвешенными ребрами (который не должен содержать циклов) из формулировки задачи планирования работ, а затем с помощью класса `AcyclicLP` (см. утверждение Ф) находит дерево самых длинных путей и выводит длины найденных путей, которые как раз равны моментам начала каждой работы.

```
% more jobsPC.txt
10
41.0  1 7 9
51.0  2
50.0
36.0
38.0
45.0
21.0  3 8
32.0  3 8
32.0  2
29.0  4 6
```

```
% java CPM < jobsPC.txt
Моменты начала:
   0:   0.0
   1:  41.0
   2: 123.0
   3:  91.0
   4:  70.0
   5:   0.0
   6:  70.0
   7:  41.0
   8:  91.0
   9:  41.0
Момент завершения: 173.0
```

Утверждение Х. Метод критического пути решает задачу планирования параллельных работ с отношениями предшествования за линейное время.

Доказательство. Почему работает метод критического пути? Корректность алгоритма основывается на двух фактах. Во-первых, каждый путь в ориентированном ациклическом графе представляет собой последовательность моментов начала и завершения работ, которые разделены отношениями предшествования с нулевыми весами: длина любого пути от источника s до любой вершины v в графе равна нижней границе момента начала/завершения, представленного вершиной v, т.к. здесь придется просто планировать эти работы одну за другой на одном и том же процессоре. В частности, длина самого длинного пути из s до стока t равна *нижней* границе времени завершения всех работ. Во-вторых, все моменты начала и конца, определяемые самыми длинными путями, *можно вычислить*: каждая работа начинается после завершения всех работ, для которых она является преемником в ограничении предшествования, т.к. момент начала равен длине *самого длинного* пути от источника до нее. В частности, длина самого длинного пути от s до t равна *верхней* границе моментов завершения всех работ. Линейная производительность непосредственно следует из утверждения Ф.

Работа	*Время*	*Относительно чего*
2	12.0	4
2	70.0	7
4	80.0	0

Рис. 4.4.22. *Добавление предельных сроков в планирование работ*

Исходная задача

работа	*начало*
0	0.0
1	41.0
2	123.0
3	91.0
4	70.0
5	0.0
6	70.0
7	41.0
8	91.0
9	41.0

2 через 12.0 после 4

работа	*начало*
0	0.0
1	41.0
2	123.0
3	91.0
4	**111.0**
5	0.0
6	70.0
7	41.0
8	91.0
9	41.0

2 через 70.0 после 7

работа	*начало*
0	0.0
1	41.0
2	123.0
3	91.0
4	111.0
5	0.0
6	70.0
7	**53.0**
8	91.0
9	41.0

4 через 80.0 после 0

невозможно!

Рис. 4.4.23. *Относительные предельные сроки в планировании работ*

Планирование параллельных работ с относительными предельными сроками

Условные предельные сроки могут задаваться относительно момента начала самой первой работы. Предположим, что в задаче планирования работ имеется еще один тип ограничения, который требует, чтобы некоторая работа обязательно началась до истечения заданного времени относительно момента начала другой работы. Такие ограничения обычно возникают в критичных по времени производственных процессах и многих других областях, и они могут существенно затруднить решение задачи планирования работ. Например (рис. 4.4.22–4.4.24), предположим, что в наш пример нужно добавить ограничение, что работа 2 должна начаться не позднее 12 единиц времени после начала работы 4. На самом деле этот предельный срок является ограничением на момент начала работы 4: она должна начаться не раньше 12 единиц времени до момента начала работы 2. В графике для нашего примера есть место для этого предельного срока: начало работы 4 можно перенести в момент 111 — за 12 единиц времени до запланированного начала работы 2. Учтите, что при большой продолжительности работы 4 это изменение увеличило бы время окончания всего графика. Аналогично, если добавить в расписание предельный срок начала работы 2 не позже 70 единиц времени после начала работы 7, то в графике имеется место, чтобы сдвинуть начало работы 7 в момент 53, не сдвигая работы 3 и 8. Но если добавить ограничение, что работа 4 должна начаться не позже 80 единиц времени после работы 0, то такое расписание *невозможно*. Условия, что работа 4 должна начаться не позже 80 единиц времени после работы 0, а работа 2 должна начаться не позже 12 единиц после работы 4, означают, что работа 2 должна начаться не позже 93 единиц времени после работы 0, но работа 2 должна начаться не раньше 123 единиц времени после работы 0 — из-за цепочки 0 (41 единица времени) перед 9 (29 единиц времени) перед 6 (21 единица времени) перед 8 (32 единицы времени) перед 2. Понятно, что добавление предельных сроков множит возможности и существенно усложняет первоначально легкую задачу.

Утверждение Ц. Планирование параллельных работ с относительными предельными сроками представляет собой задачу нахождения кратчайших путей в орграфах с взвешенными ребрами (с возможными циклами и отрицательными весами ребер).

Доказательство. Можно использовать то же построение, что и для утверждения X, но добавляя по одному ребру для каждого предельного срока: если работа v должна начаться в пределах d единиц времени от начала работы w,

то нужно добавить ребро из v до w с *отрицательным* весом d. Затем полученная задача преобразуется в задачу нахождения кратчайших путей, изменив знаки всех весов в орграфе. Доказательство корректности остается тем же, *если это расписание возможно*. Как мы увидим, определение возможности расписания вносит свой вклад в сложность вычислений.

Этот пример показывает, что отрицательные веса могут играть важную роль в моделях практических приложений. Если можно найти эффективное решение задачи поиска кратчайших путей с отрицательными весами, то можно найти и эффективное решение задачи планирования параллельных работ с относительными предельными сроками. Ни один из рассмотренных нами алгоритмов не может сделать это: для работы алгоритма Дейкстры обязательно требование неотрицательных весов, а в алгоритме 4.10 нужно, чтобы орграф не содержал циклов. Так что теперь мы займемся работой с орграфами, в которых допустимы и отрицательные веса ребер, и циклы.

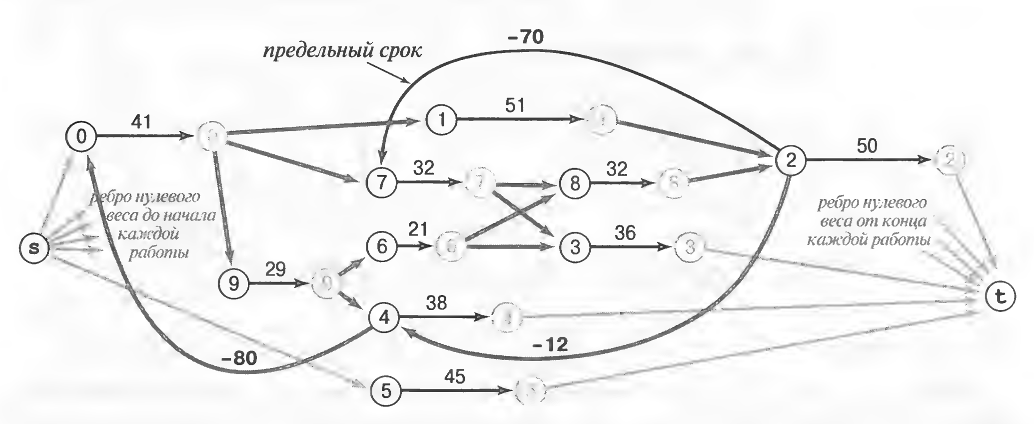
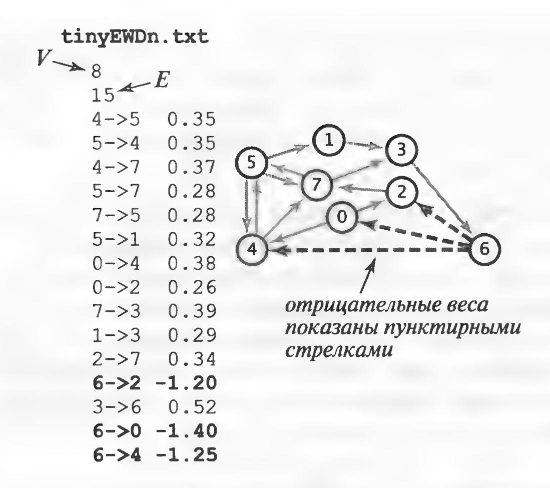


Рис. 4.4.24. Орграф с взвешенными ребрами, представляющий задачу параллельного планирования с ограничениями предшествования и относительными предельными сроками

Кратчайшие пути в орграфах с взвешенными ребрами общего вида

Только что рассмотренный пример планирования работ с предельными сроками показывает, что отрицательные веса — не просто математические странности. Наоборот, они существенно расширяют применимость задачи поиска кратчайших путей в качестве модели для решения задач. Поэтому сейчас мы рассмотрим алгоритмы для орграфов с взвешенными ребрами, которые могут содержать *и* циклы, *и* отрицательные веса. Однако сначала мы познакомимся с некоторыми основными свойствами таких орграфов, чтобы лучше разобраться в кратчайших путях. На рис. 4.4.25 приведен небольшой пример, демонстрирующий эффекты разрешения отрицательных весов в кратчайших путях орграфов (см. также рис. 4.4.27). Пожалуй, наиболее важно то, что при наличии отрицательных весов кратчайшие пути могут содержать *больше* ребер, чем пути с большим весом. В случае положительных весов мы старались спрямлять пути, но при наличии отрицательных весов мы ищем *обходные пути*, содержащие ребра с отрицательными весами. Это поведение совершенно сводит на нет наше интуитивное понимание поиска “кратчайших” путей, поэтому придется подавлять его и рассматривать задачу на чисто абстрактном уровне.



Дерево кратчайших путей из 0

	edgeTo[]	distTo[]
0		
1	5->1	0.93
2	0->2	0.26
3	7->3	0.99
4	6->4	0.26
5	4->5	0.61
6	3->6	1.51
7	2->7	0.60

Рис. 4.4.25. Орграф с отрицательными весами ребер

Заблуждение I

Первая мысль, которая приходит на ум — нужно найти ребро с наименьшим (самым отрицательным) весом и добавить абсолютную величину этого веса ко всем весам ребер, чтобы преобразовать орграф в вид с неотрицательными весами. Этот примитивный способ вообще неработоспособен, т.к. кратчайший путь в новом орграфе слабо связан с кратчайшими путями в старом. Чем больше ребер в пути, тем более он пострадает от такого преобразования (см. упражнение 4.4.14).

Заблуждение II

Вторая мысль — попробовать как-то адаптировать алгоритм Дейкстры. Фундаментальная сложность этого подхода в том, что алгоритм основан на просмотре вершин в порядке возрастания их расстояния от источника. В доказательстве корректности алгоритма в утверждении Т предполагается, что добавление ребра к пути удлиняет этот путь. Но любое ребро с отрицательным весом *укорачивает* путь, поэтому такое предположение не обязательно верно (см. упражнение 4.4.14).

Отрицательные циклы

При изучении графов, которые могут содержать ребра отрицательного веса, концепция кратчайшего пути теряет смысл при наличии в орграфе цикла с отрицательным весом. К примеру, рассмотрим орграф, изображенный на рис. 4.4.26: он идентичен нашему первому примеру, только ребро 5->4 имеет вес –0.66. Тогда вес цикла 4->7->5->4 равен

$$0.37 + 0.28 - 0.66 = -0.01$$

Можно крутиться по этому циклу, генерируя все более короткие пути! Не обязательно все ребра ориентированного цикла должны иметь отрицательные веса — важна лишь *сумма* весов составляющих его ребер.

Определение. *Отрицательный цикл* в орграфе с взвешенными ребрами — это ориентированный цикл, общий вес которого (сумма весов его ребер) отрицателен.

Предположим, что некоторая вершина на пути от s до достижимой вершины v тоже находится в отрицательном цикле. В этом случае существование кратчайшего пути от s до v теряет смысл, т.к. с помощью цикла можно построить путь с весом, меньшим любого заданного значения. То есть при наличии отрицательных циклов задача поиска кратчайших путей может оказаться некорректной.

Утверждение Ч. Кратчайший путь из вершины s до вершины v в орграфе с взвешенными ребрами существует тогда и только тогда, когда существует хотя бы один ориентированный путь из s до v *и* ни одна вершина ни одного такого пути из s до v не проходит через отрицательный цикл.

Доказательство. См. вышеприведенное рассуждение и упражнение 4.4.29.

Из требования отсутствия в кратчайших путях вершин из отрицательных циклов следует, что кратчайшие пути могут быть только простыми и что для таких вершин можно вычислить дерево кратчайших путей, как это можно сделать для ребер с положительными весами.

Заблуждение III

Имеются отрицательные циклы или нет, все равно существует кратчайший *простой* путь, соединяющий источник с каждой вершиной, достижимой из этого источника. Почему не определить кратчайшие пути таким образом? К сожалению, наилучший известный алгоритм для решения этой задачи требует в худшем случае экспоненциального времени (см. главу 6). Обычно мы считаем такие задачи "слишком сложными для решения" и изучаем более простые варианты.

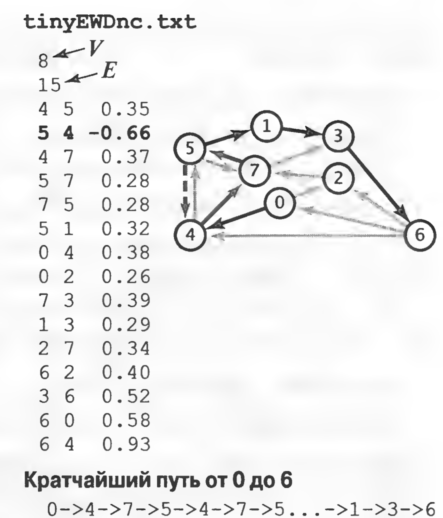


Рис. 4.4.26. Орграф с взвешенными ребрами с отрицательным циклом

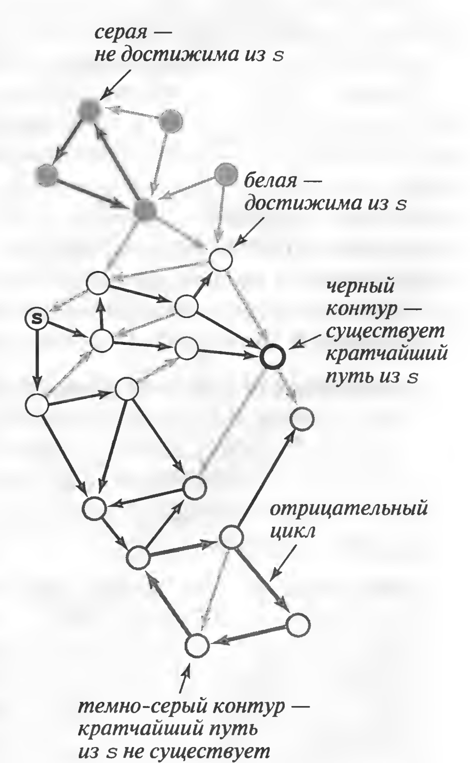


Рис. 4.4.27. Возможные варианты при поиске кратчайших путей

Итак, корректно поставленный и разрешимый вариант задачи поиска кратчайших путей в орграфах с взвешенными ребрами требует, чтобы:

- вершинам, не достижимым из источника, присваивался вес кратчайшего пути $+\infty$;
- вершинам, которые находятся на пути из источника, содержащем отрицательный цикл, присваивался вес кратчайшего пути $-\infty$;
- для всех остальных вершин вычислялся вес (и дерево) кратчайших путей.

На протяжении этого раздела мы накладывали ограничения на задачу поиска кратчайших путей, чтобы разработать алгоритмы для ее решения. Вначале мы запретили отрицательные веса, потом запретили отрицательные циклы. Теперь мы зададим менее строгие ограничения, и будем рассматривать следующие задачи в орграфах общего вида.

Обнаружение отрицательных циклов. Содержит ли заданный орграф с взвешенными ребрами отрицательный цикл? Если да, нужно найти один такой цикл.

Кратчайшие пути из одного источника, если отрицательные циклы недостижимы. Пусть задан орграф с взвешенными ребрами с источником `s` и без отрицательных циклов, достижимых из `s`. Необходимо отвечать на вопросы вида *Существует ли ориентированный путь из* `s` *до заданной вершины* `v`? Если да, нужно найти *кратчайший* такой путь (с минимальным общим весом).

Итак, кратчайшие пути в орграфах с отрицательными циклами — некорректная задача, и в таких орграфах *невозможно* эффективно решить задачу поиска простых кратчайших путей. Но в практических ситуациях *можно* обнаружить отрицательные циклы. Например, в задаче планирования работ с предельными сроками можно считать, что отрицательные циклы будут встречаться относительно редко: ограничения и сроки вытекают из логических реальных ограничений, поэтому любые отрицательные циклы, скорее всего, означают ошибку в формулировке задачи. Тогда разумно будет найти отрицательные циклы, исправить ошибки и затем вычислить расписание в задаче без отрицательных циклов. То есть цель вычисления — обнаружение отрицательного цикла. Способ, разработанный Беллманом (Bellman) и Фордом (Ford) в конце 1950-х годов, обеспечивает простой и эффективный фундамент для решения обеих этих задач, а, кроме того, эффективен и для орграфов с положительными весами.

Утверждение Ш (алгоритм Беллмана-Форда). Следующий метод решает задачу кратчайших путей из одного заданного источника в любом орграфе с взвешенными ребрами и V вершинами, в котором нет отрицательных циклов, достижимых из `s`: в `distTo[s]` заносится 0, а во все остальные элементы `distTo[]` — бесконечность. Затем выполняется релаксация всех ребер в любом порядке, и выполняется V таких проходов.

Доказательство. Для любой вершины `t`, достижимой из `s`, рассмотрим конкретный кратчайший путь v_0->v_1->...->v_k, где v_0 = `s` и v_k = `t`. В силу отсутствия отрицательных циклов такой путь существует, и `k` не может быть больше $V-1$. Докажем по индукции по `i`, что после `i`-го прохода алгоритм вычисляет кратчайший путь от `s` до v_i. Базовый случай (`i` = 0) тривиален. Предположим, что для некоторого `i` утверждение верно: v_0->v_1->...->v_i — кратчайший путь от `s` до v_i, и его длина равна `distTo[`v_i`]`. Теперь выполним на `i`-м проходе релаксацию каждой вершины, включая v_i, тогда `distTo[`v_i`+1]` не больше `distTo[`v_i`]` плюс вес ребра v_i->v_{i+1}. Тогда после `i`-го прохода значение `distTo[`v_{i+1}`]` должно быть

равно distTo[v_i] плюс вес v_i->v_{i+1}. Оно не может быть больше, т.к. на i-м проходе выполняется релаксация каждой вершины и, в частности, v_i, и оно не может быть меньше, т.к. это длина v_0->v_1->...->v_{i+1} — кратчайшего пути. Значит, после (i+1)-го прохода алгоритм вычисляет кратчайший путь от s до v_{i+1}.

Утверждение Ч (продолжение). Для работы алгоритма Беллмана-Форда необходимо время, пропорциональное *EV*, и дополнительная память, пропорциональная *V*.

Доказательство. Каждый из *V* проходов выполняет релаксацию *E* ребер.

Этот метод является весьма общим, поскольку в нем не задается порядок релаксации ребер. Но мы сейчас рассмотрим менее общий метод, где всегда выполняется релаксация всех ребер, направленных из любой вершины (в любом порядке). Простоту такого подхода демонстрирует следующий код:

```
for (int pass = 0; pass < G.V(); pass++)
   for (v = 0; v < G.V(); v++)
      for (DirectedEdge e : G.adj(v))
         relax(e);
```

Мы не будем подробно рассматривать этот вариант, поскольку он *всегда* выполняет релаксацию *VE* ребер, но простая модификация делает алгоритм гораздо более эффективным в обычных случаях.

Алгоритм Беллмана-Форда на основе очереди

Нетрудно *заранее* определить, что многочисленные ребра не приведут к успешной релаксации на любом проходе: единственные ребра, которые приводят к изменению distTo[] — это направленные из вершины, значение distTo[] которой изменилось на предыдущем проходе. Для отслеживания таких вершин можно использовать очередь. Работа алгоритма для нашего стандартного примера с положительными весами показана на рис. 4.4.28. Слева на этом рисунке приведены элементы очереди на каждом проходе (жир-

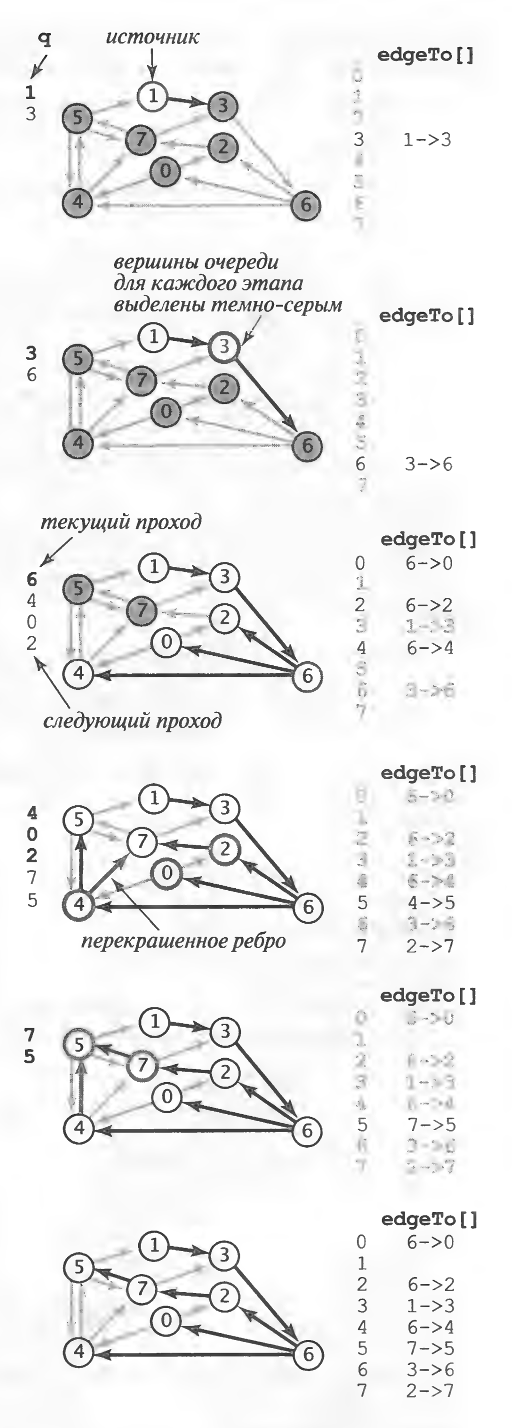


Рис. 4.4.28. Трассировка алгоритма Беллмана-Форда

ные цифры), а за ними элементы очереди для следующего прохода (обычные цифры). Вначале в очередь заносится источник, а затем вычисляется ДКП следующим образом.

- Выполняется релаксация ребра 1->3, а вершина 3 помещается в очередь.
- Выполняется релаксация ребра 3->6, а вершина 6 помещается в очередь.
- Выполняется релаксация ребер 6->4, 6->0 и 6->2, а вершины 4, 0 и 2 помещаются в очередь.
- Выполняется релаксация ребер 4->7 и 4->5, а вершины 7 и 4 помещаются в очередь. Потом выполняется релаксация непригодных ребер 0->4 и 0->2. Затем выполняется релаксация 2->7 (и перекрашивание 4->7).
- Выполняется релаксация ребра 7->5 (и перекрашивание 4->5), но вершина 5 не помещается в очередь, т.к. она уже там. Потом выполняется релаксация непригодного ребра 7->3. Затем выполняется релаксация непригодных ребер 5->1, 5->4 и 5->7, и очередь становится пустой.

Реализация

Реализация алгоритма Беллмана-Форда в соответствии с этим описанием требует на удивление мало кода — см. листинг 4.4.11. Она основана на двух дополнительных структурах данных:

- очередь q вершин, ожидающих релаксации;
- индексированный вершинами логический массив onQ[], который указывает, какие вершины занесены в очередь, чтобы не было повторений.

Листинг 4.4.11. Алгоритм 4.11. Алгоритм Беллмана–Форда (на основе очереди)

```
public class   BellmanFordSP
{
   private double[] distTo;         // длина пути до v
   private DirectedEdge[] edgeTo;  // последнее ребро на пути до v
   private boolean[] onQ;          // находится ли данная вершина в очереди?
   private Queue<Integer> queue;   // вершины, для которых выполняется релаксация
   private int cost;               // количество вызовов relax()
   private Iterable<DirectedEdge> cycle;  // отрицательный цикл в edgeTo[]?
   public BellmanFordSP(EdgeWeightedDigraph G, int s)
   {
      distTo = new double[G.V()];
      edgeTo = new DirectedEdge[G.V()];
      onQ = new boolean[G.V()];
      queue = new Queue<Integer>();
      for (int v = 0; v < G.V(); v++)
         distTo[v] = Double.POSITIVE_INFINITY;
      distTo[s] = 0.0;
      queue.enqueue(s);
      onQ[s] = true;
      while (!queue.isEmpty() && !this.hasNegativeCycle())
      {
         int v = queue.dequeue();
         onQ[v] = false;
         relax(v);
      }
   }

   private void relax(int v)
```

```
  // См. листинг 4.4.12.
  public double distTo(int v)        // Стандартные методы клиентских запросов
  public boolean hasPathTo(int v)    //   для реализаций ДКП
  public Iterable<Edge> pathTo(int v) //   (см. листинг 4.4.6)

  private void findNegativeCycle()
  public boolean hasNegativeCycle()
  public Iterable<Edge> negativeCycle()
  // См. листинг 4.4.13.
}
```

В данной реализации алгоритма Беллмана-Форда используется вариант метода relax(), который помещает в очередь (не допуская повторений) вершины, указываемые ребрами, для которых успешно выполнена релаксация, и периодически проверяет наличие отрицательного цикла в массиве edgeTo[] (см. текст).

Вначале в очередь заносится источник s, а затем в цикле выполняется выборка вершин из очереди и их релаксация. Для занесения вершин в очередь мы немного расширили реализацию метода relax() из листинга 4.4.4, чтобы помещать в очередь вершину, в которую направлено любое ребро, если для него успешно выполнена релаксация (см. листинг 4.4.12). Структуры данных обеспечивают, что

- в очереди может находиться только одна копия каждой вершины;
- каждая вершина, значения edgeTo[] и distTo[] изменились на каком-то проходе, обрабатываются на следующем проходе.

Листинг 4.4.12. Релаксация для алгоритма Беллмана-Форда

```
private void relax(EdgeWeightedDigraph G, int v)
{
   for (DirectedEdge e : G.adj(v))
   {
      int w = e.to();
      if (distTo[w] > distTo[v] + e.weight())
      {
         distTo[w] = distTo[v] + e.weight();
         edgeTo[w] = e;
         if (!onQ[w])
         {
            q.enqueue(w);
            onQ[w] = true;
         }
      }
      if (cost++ % G.V() == 0)
         findNegativeCycle();
   }
}
```

Чтобы завершить реализацию, необходимо гарантировать, что алгоритм завершается после V проходов. Можно просто явно подсчитывать проходы. Но в нашей реализации BellmanFordSP в алгоритме 4.11 применяется другой подход, который мы подробно рассмотрим ниже, в разделе "Обнаружение отрицательных циклов": выполняется проверка на наличие отрицательных циклов в подмножестве ребер орграфа из массива edgeTo[], и при обнаружении такого цикла работа прекращается.

20 Зак. 3239

Утверждение Щ. Для любого орграфа с V вершинами и E взвешенными ребрами реализация алгоритма Беллмана-Форда на основе очереди решает задачу поиска кратчайших путей из заданного источника s (или находит отрицательный цикл, достижимый из s), требуя в худшем случае время, пропорциональное EV, и объем дополнительной памяти, пропорциональный V.

Доказательство. Если существует отрицательный цикл, достижимый из s, то алгоритм завершает работу после релаксаций, соответствующих ($V-1$)-му проходу обобщенного алгоритма, который описан в утверждении Ш (т.к. все кратчайшие пути содержат меньше $V-1$ ребер). При наличии отрицательного цикла, достижимого из s, очередь никогда не станет пустой. После выполнения релаксаций, соответствующих V-му проходу обобщенного алгоритма из утверждения Ш массив edgeTo[] содержит путь с циклом (соединяющим некоторую вершину w с самой собой), и этот цикл может быть отрицательным, т.к. для повторного включения в путь вершины w путь от s до второго вхождения w должен быть короче пути от s до первого вхождения w. В худшем случае алгоритм действует аналогично обобщенному алгоритму и выполняет релаксацию всех E ребер на каждом из V проходов.

Алгоритм Беллмана-Форда на основе очереди — эффективный метод для решения задачи поиска кратчайших путей, и он широко применяется на практике, даже для случаев, когда веса ребер положительны. Например, как показано на рис. 4.4.29, наш пример с 250 вершинами решается за 14 проходов и выполняет меньше сравнений длин путей, чем алгоритм Дейкстры для этой же задачи.

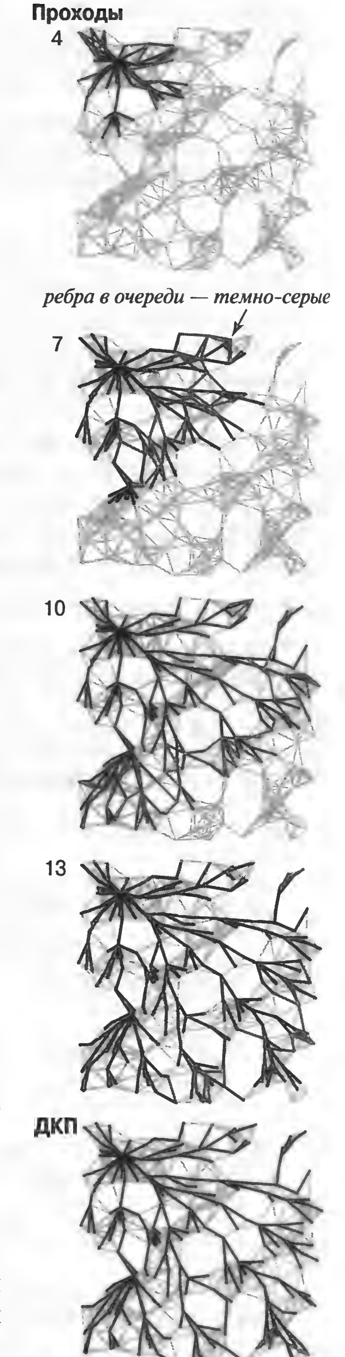


Рис. 4.4.29. Работа алгоритма Беллмана-Форда (250 вершин)

Отрицательные веса

На рис. 4.4.30 показана работа алгоритма Беллмана-Форда на орграфе с отрицательными весами. Работа начинается с помещения источника в очередь q, после чего вычисляется ДКП следующим образом.

- Выполняется релаксация ребер 0->2 и 0->4, и вершины 2 и 4 помещаются в очередь.
- Выполняется релаксация ребра 2->7, и вершина 7 помещается в очередь. Потом выполняется релаксация ребра 4->5, и вершина 5 помещается в очередь. Затем выполняется релаксация ребра 4->7, которое непригодно.
- Выполняется релаксация ребер 7->3 и 5->1, и вершины 3 и 1 помещаются в очередь. Затем выполняется релаксация ребер 5->4 и 5->7, которые непригодны.

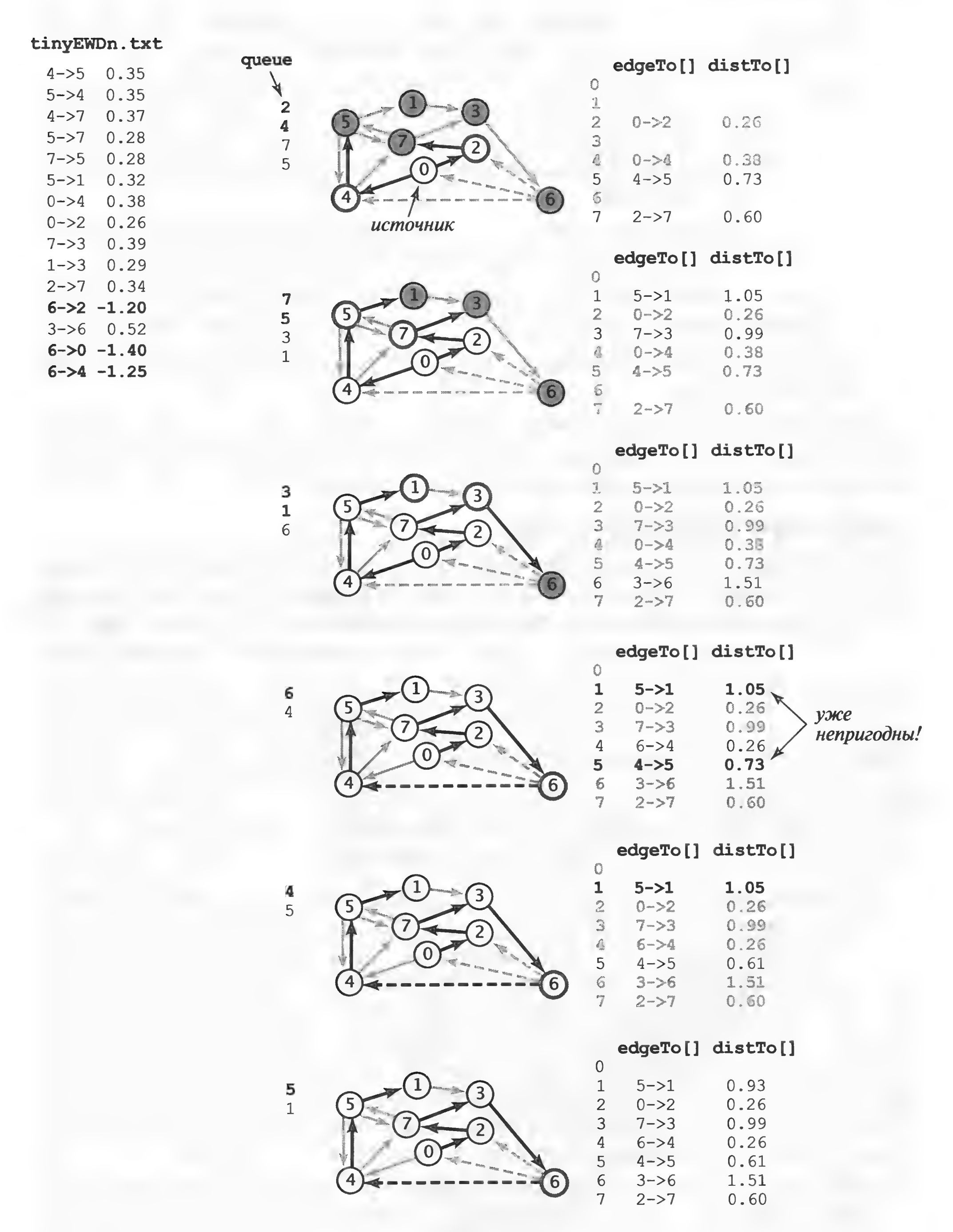


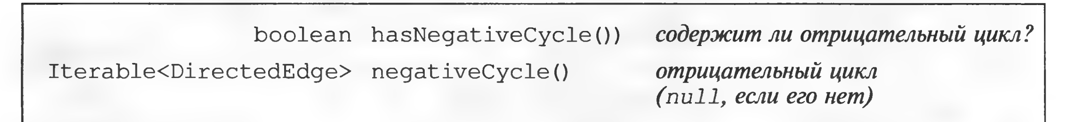
Рис. 4.4.30. *Трассировка работы алгоритма Беллмана-Форда (с отрицательными весами)*

- Выполняется релаксация ребра 3->6, и вершина 6 помещается в очередь. Затем выполняется релаксация ребра 1->3, которое непригодно.
- Выполняется релаксация ребра 6->4, и вершина 4 помещается в очередь. Это ребро отрицательного веса дает более короткий путь до вершины 4, поэтому для его ребер нужно выполнить повторную релаксацию (в первый раз это было на проходе 2). После этого расстояния до вершин 5 и 1 уже не верны и будут исправлены на последующих проходах.
- Выполняется релаксация ребра 4->5, и вершина 5 помещается в очередь. Затем выполняется релаксация ребра 4->7, которое все так же непригодно.
- Выполняется релаксация ребра 5->1, и вершина 1 помещается в очередь. Затем выполняется релаксация ребер 5->4 и 5->7, которые все так же непригодны.
- Выполняется релаксация ребра 1->3, которое все так же непригодно, и очередь становится пустой.

Дерево кратчайших путей в данном примере — один длинный путь от 0 до 1. Для ребер из вершин 4, 5 и 1 релаксация выполнялась дважды. Чтобы лучше разобраться в этом примере, просмотрите еще раз доказательство утверждения Ш.

Обнаружение отрицательных циклов

Наша реализация BellmanFordSP проверяет наличие отрицательных циклов, чтобы избежать бесконечного цикла. Код, выполняющий эту проверку, можно применить и для того, чтобы позволить клиентам обнаруживать и извлекать отрицательные циклы. Для этого в API SP (см. рис. 4.4.7) нужно добавить методы, представленные на рис. 4.4.31.



boolean	hasNegativeCycle())	*содержит ли отрицательный цикл?*
Iterable<DirectedEdge>	negativeCycle()	*отрицательный цикл (null, если его нет)*

Рис. 4.4.31. *Расширения API поиска кратчайших путей для работы с отрицательными циклами*

Реализация этих методов не сложна и приведена в листинге 4.4.13. После отработки конструктора BellmanFordSP на основании доказательства утверждения Щ можно сказать, что в орграфе существует отрицательный цикл, достижимый из источника в том и только том случае, если после V-го прохода по всем ребрам очередь остается непустой. Более того, отрицательный цикл должен содержаться в подграфе ребер, содержащихся в массиве edgeTo[]. Значит, для реализации метода negativeCycle() нужно составить орграф с взвешенными ребрами из ребер edgeTo[] и искать цикл в этом орграфе. Для поиска цикла используется вариант реализации DirectedCycle из раздела 4.3, адаптированный для работы с орграфами с взвешенными ребрами (см. упражнение 4.4.12). Стоимость такой проверки амортизируется:

- добавлением переменной экземпляров cycle и приватного метода findNegativeCycle(), который заносит в переменную cycle итератор для ребер отрицательного цикла, если он обнаружен (и null, если не обнаружен);
- вызовом findNegativeCycle() в каждом V-м вызове relax().

Листинг 4.4.13. Методы обнаружения отрицательного цикла для алгоритма Беллмана-Форда

```
private void findNegativeCycle()
{
   int V = edgeTo.length;
   EdgeWeightedDigraph spt;
   spt = new EdgeWeightedDigraph(V);
   for (int v = 0; v < V; v++)
      if (edgeTo[v] != null)
         spt.addEdge(edgeTo[v]);
   EdgeWeightedCycleFinder cf;
   cf = new EdgeWeightedCycleFinder(spt);
   cycle = cf.cycle();
}

public boolean hasNegativeCycle()
{  return cycle != null;  }

public Iterable<Edge> negativeCycle()
{  return cycle;  }
```

При таком подходе цикл в конструкторе гарантированно завершается. Более того, клиенты могут вызывать метод `hasNegativeCycle()`, чтобы узнать, существует ли отрицательный цикл, достижимый из источника (и `negativeCycle()`, чтобы получить этот цикл). Добавить возможность обнаружения любого отрицательного цикла в орграфе также несложно (см. упражнение 4.4.43).

На рис. 4.4.32 показано выполнение алгоритма Беллмана-Форда в орграфе с отрицательным циклом. Первые два прохода совпадают с проходами по графу `tinyEWDn.txt`. На третьем проходе, после релаксации ребер `7->3` и `5->1` и помещения в очередь вершин `3` и `1`, выполняется релаксация ребра с отрицательным весом `5->4`. *Эта релаксация обнаруживает отрицательный цикл* `4->5->4`. Ребро `5->4` помещается в дерево, а цикл отсекается от источника `0` в массиве `edgeTo[]`. Начиная с этого момента, алгоритм зацикливается, уменьшая расстояния до всех встречаемых вершин, пока этот цикл не обнаруживается по тому, что очередь не пуста. Цикл находится в массиве `edgeTo[]`, и его можно найти методом `findNegativeCycle()`.

Арбитраж

Рассмотрим рынок финансовых транзакций, основанный на торговле валютами. Пример можно найти в курсах валют — как в приведенном на рис. 4.4.33 файле `rates.txt`. Первая строка в этом файле содержит количество валют V; а затем в каждой строке записано название валюты и ее курсы относительно других валют. Для краткости здесь приведены только пять из сотен различных валют, торгуемых на современных валютных рынках: американские доллары (`USD`), евро (`EUR`), британские фунты (`GBP`), швейцарские франки (`CHF`) и канадские доллары (`CAD`). `t`-е число в `s`-й строке означает обменный курс — количество единиц валюты с именем в `s`-й строке, за которое можно купить 1 единицу валюты с именем в `t`-й строке. Например, из таблицы видно, что за 1000 американских долларов можно купить 741 евро.

Эта таблица эквивалентна *полному орграфу с взвешенными ребрами*, где вершины соответствуют валютам, а ребра — обменным курсам. Ребро `s->t` с весом `x` соответствует обмену `s` на `t` по цене `x`. Пути на этом орграфе задают многошаговые обмены.

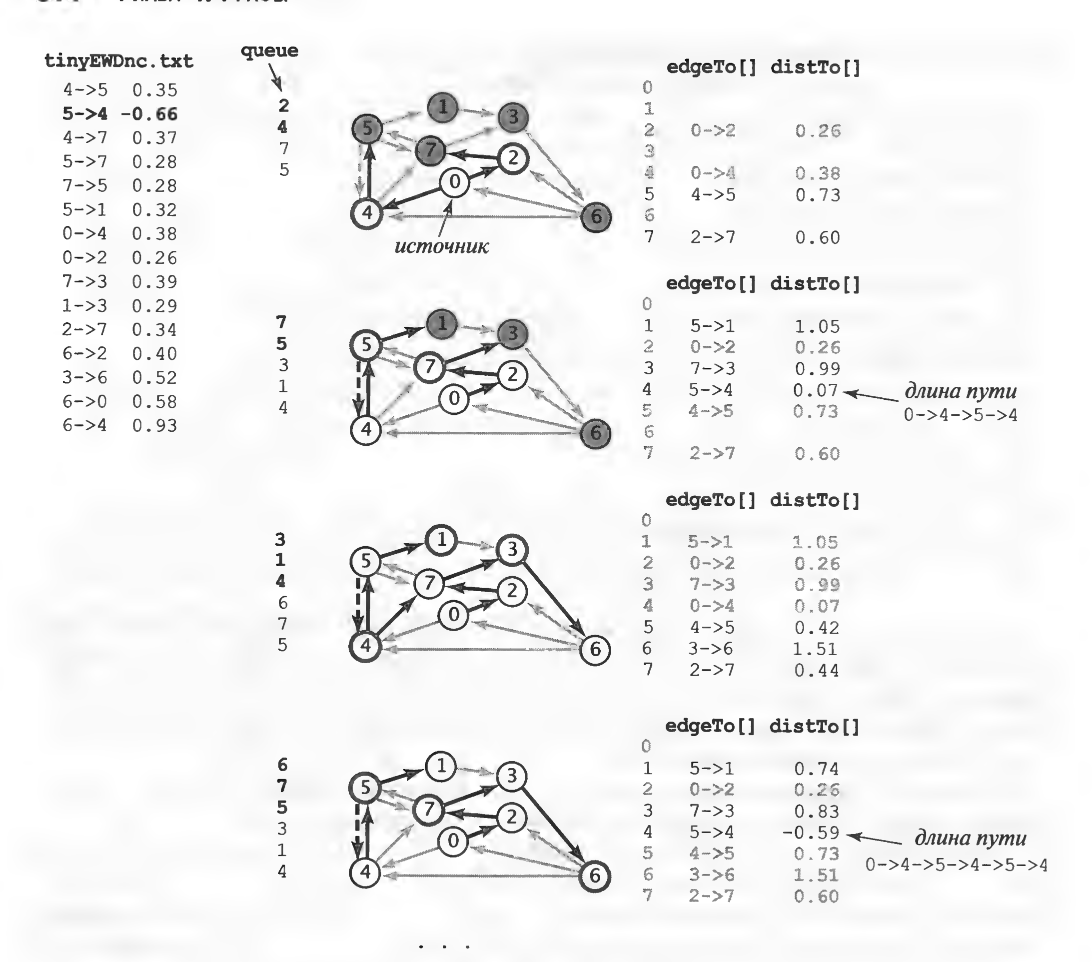


Рис. 4.4.32. Трассировка работы алгоритма Беллмана-Форда (отрицательный цикл)

```
% more rates.txt
5
USD  1      0.741  0.657  1.061  1.005
EUR  1.349  1      0.888  1.433  1.366
GBP  1.521  1.126  1      1.614  1.538
CHF  0.942  0.698  0.619  1      0.953
CAD  0.995  0.732  0.650  1.049  1
```

Рис. 4.4.33. Пример файла с курсами обмена валют

Например, сочетание вышеприведенного обмена с ребром t->u с весом y дает путь s->t->u, который представляет путь преобразования 1 единиц валюты s в xy единиц валюты u. Например, за наши евро можно купить

1012,206 = 741 × 1,366 канадских долларов.

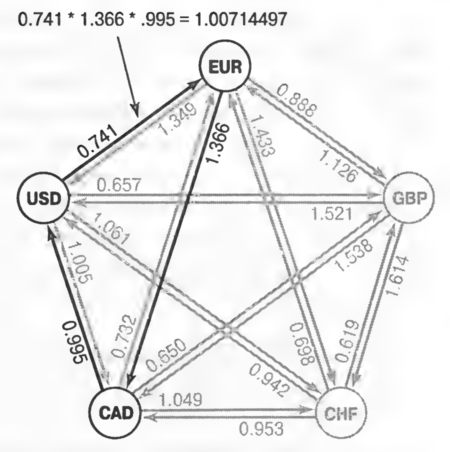


Рис. 4.4.34. Возможность арбитража

Обратите внимание, что это выгоднее непосредственного обмена американских долларов на канадские. Казалось бы, xy всегда должно быть равно весу s->u, но подобные таблицы представляют собой сложную финансовую систему, где согласованность не гарантируется. Так что обнаружение такого пути от s до u с максимальным весом определенно представляет интерес. Но еще более интересен случай, когда произведение весов ребер *меньше* веса ребра из последней вершины назад к первой. В нашем примере предположим, что вес ребра u->s равен z и что xyz>1. Тогда цикл s->t->u->s позволяет обменять 1 единицу валюты s в более чем 1 единицу (xyz) валюты s. То есть можно получить прибыль в 100(xyz - 1) процентов, поменяв s через t и u снова на s. Например, если обменять 1012,206 канадских долларов назад на американские доллары, мы получим 1012,206 × 0,995 = 1007,14497 долларов — с прибылью в 7,14497 долларов. С виду это немного, но валютный трейдер может оперировать миллионами долларов и может выполнять такие обмены каждую минуту — а это 7000 долларов в минуту или 420 000 долларов в час! Подобная ситуация представляет собой пример возможности *арбитража* (рис. 4.4.34), который позволял бы трейдерам получать неограниченные прибыли, если бы не было ограничивающих факторов за пределами этой модели, таких как комиссия за транзакции или ограничения на размеры транзакций. Но даже и с учетом этих ограничений арбитраж может быть весьма прибылен в реальном мире.

Но какое отношение имеет эта задача к кратчайшим путям? Ответ на этот вопрос довольно прост.

Утверждение Э. Задача арбитража представляет собой задачу обнаружения отрицательных циклов в орграфе с взвешенными ребрами.

Доказательство. Заменим каждый вес его логарифмом с обратным знаком. После этого вычисление весов путей с помощью перемножения весов ребер в исходной задаче соответствует их сложению в преобразованной задаче (рис. 4.4.35). А именно, любое произведение $w_1 w_2 \dots w_k$ соответствует сумме $-\ln(w_1) - \ln(w_2) - \dots - \ln(w_k)$. Полученные веса ребер могут быть положительными или отрицательными, путь от v до w означает обмен валюты v на валюту w, а любой отрицательный цикл означает возможность арбитража.

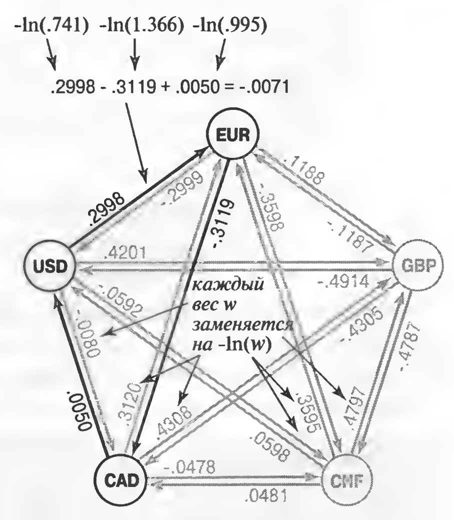


Рис. 4.4.35. Отрицательный цикл, означающий возможность арбитража

В нашем примере возможны все транзакции, т.е. орграф является полным, и поэтому любой отрицательный цикл достижим из любой вершины. В реальных ситуациях некоторые ребра могут отсутствовать, поэтому нужен конструктор копировании с одним аргументом, описанный в упражнении 4.4.43. Не найден эффективный алгоритм для нахождения *наилучшей* возможности арбитража (цикл в орграфе с наиболее отрицательным весом), да и граф не может быть слишком большим из-за огромной вычислительной сложности, но в принципе достаточно быстрого алгоритма для нахождения *любой* возможности арбитража. Трейдер, вооруженный таким алгоритмом, может пропустить множество подобных возможностей, прежде чем доступный алгоритм обнаружит какую-нибудь из них.

Листинг 4.4.14. Арбитраж при обмене валют

```
public class  Arbitrage
{
   public static void main(String[] args)
   {
      int V = StdIn.readInt();
      String[] name = new String[V];
      EdgeWeightedDigraph G = new EdgeWeightedDigraph(V);
      for (int v = 0; v < V; v++)
      {
         name[v] = StdIn.readString();
         for (int w = 0; w < V; w++)
         {
            double rate = StdIn.readDouble();
            DirectedEdge e = new DirectedEdge(v, w, -Math.log(rate));
            G.addEdge(e);
         }
      }

      BellmanFordSP spt = new BellmanFordSP(G, 0);
      if (spt.hasNegativeCycle())
      {
         double stake = 1000.0;
         for (DirectedEdge e : spt.negativeCycle())
         {
            StdOut.printf("%10.5f %s ", stake, name[e.from()]);
            stake *= Math.exp(-e.weight());
            StdOut.printf("= %10.5f %s\n", stake, name[e.to()]);
         }
      }
      else StdOut.println("Нет возможности арбитража");
   }
}
```

Этот клиент класса `BellmanFordSP` находит возможность арбитража в таблице курсов валют. Для этого он строит представление таблицы курсов в виде полного графа, а затем использует алгоритм Беллмана-Форда, чтобы найти отрицательный цикл в полученном орграфе.

```
% java Arbitrage < rates.txt
1000.00000 USD =  741.00000 EUR
 741.00000 EUR = 1012.20600 CAD
1012.20600 CAD = 1007.14497 USD
```

Преобразование в доказательстве утверждения Э полезно не только в случае арбитража, т.к. оно сводит обмен валют к задаче нахождения кратчайших путей. В силу монотонности функции логарифма (а мы изменяем его знак) произведение максимально как раз тогда, когда сумма этих логарифмов минимальна. Веса ребер могут быть положительными или отрицательными — в любом случае кратчайший путь от v до w означает наилучший способ конвертации валюты v в валюту w.

Перспектива

В табл. 4.4.2 приведена сводка важных характеристик алгоритмов поиска кратчайших путей, которые были рассмотрены в данном разделе. Основная причина выбора среди этих алгоритмов связана с базовыми свойствами имеющегося орграфа. Есть ли в нем отрицательные веса? Содержит ли он циклы? Есть ли среди этих циклов отрицательные? Кроме этих характеристик, могут сильно отличаться и другие характеристики орграфов с взвешенными ребрами, поэтому выбор нужного алгоритма требует наличия некоторого опыта.

Таблица 4.4.2. Характеристики производительности алгоритмов поиска кратчайших путей

Алгоритм	Ограничение	Сравнения (порядок длин путей роста)		Объем памяти	Преимущество
		Типичный	Худший случай		
Дейкстры (“энергичный”)	Положительные веса ребер	$E \log V$	$E \log V$	V	Гарантированная производительность в худшем случае
Топологическая сортировка	Ориентированные ациклические графы с взвешенными ребрами	$E + V$	$E + V$	V	Оптимален для ациклических графов
Беллмана-Форда (на основе очереди)	Без отрицательных циклов	$E + V$	VE	V	Широко применим

Исторические сведения

Задачи поиска кратчайших путей интенсивно изучались и широко применялись, начиная с 1950-х годов. История алгоритма Дейкстры для вычисления кратчайших путей похожа на историю алгоритма Прима для вычисления МОД (и связана с ней). Название *алгоритм Дейкстры* обычно широко применяется и для абстрактного метода построения ДКП с помощью добавления вершин в порядке их расстояния от источника, и для реализации в качестве оптимального алгоритма при представлении матрицей смежности, т.к. Дейкстра описал в своей статье в 1959 г. и то, и другое (а также показал, что этот же подход может вычислить МОД). Способы повышения производительности для разреженных графов зависят от последующих усовершенствований в реализациях очередей с приоритетами, которые не ориентированы конкретно на задачу поиска кратчайших путей. Повышение производительности алгоритма Дейкстры — одно из наиболее важных применений данной технологии (например, структура, которая называется *пирамида Фибоначчи*, позволяет улучшить границу для худшего случая до $E + V \log V$).

Алгоритм Беллмана-Форда доказал на практике свою полезность и нашел много областей применения, особенно для орграфов с взвешенными ребрами общего вида. В типичных приложениях время его работы обычно линейно, но в худшем случае оно равно *VE*. Разработка алгоритма поиска кратчайших путей с линейным временем выполнения в худшем случае для разреженных графов остается нерешенной. Базовый алгоритм Беллмана-Форда был разработан в 1950-х годах Фордом (L. Ford) и Беллманом (R. Bellman). Несмотря на впечатляющие рывки в производительности, которые мы наблюдаем для многих других задач обработки графов, мы еще не видели алгоритмов с лучшей производительностью в худшем случае для орграфов с отрицательными весами ребер (но без отрицательных циклов).

Вопросы и ответы

Вопрос. **Почему мы рассматриваем отдельные типы данных для неориентированных графов, ориентированных графов, неориентированных графов с взвешенными ребрами и взвешенных орграфов?**

Ответ. Это сделано как для упрощения клиентского кода, так и для повышения понятности и эффективности кода реализации в невзвешенных графах. В приложениях или системах, где необходимо обрабатывать все виды графов, несложно определить АТД, от которого можно породить АТД `Graph` для невзвешенных неориентированных графов (раздел 4.1), `Graph` для невзвешенных орграфов (раздел 4.2), `EdgeWeightedGraph` для неориентированных графов с взвешенными ребрами (раздел 4.3) или `EdgeWeightedDigraph` для ориентированных графов с взвешенными ребрами из данного раздела.

Вопрос. **Как можно находить кратчайшие пути в неориентированных (но взвешенных) графах?**

Ответ. Для положительных весов ребер годится алгоритм Дейкстры. Нужно просто построить объект `EdgeWeightedDigraph`, соответствующий исходному графу, в котором каждое неориентированное ребро представлено двумя ориентированными ребрами (по одному в каждом направлении), а потом применить алгоритм Дейкстры. Если веса ребер могут быть отрицательными, то для таких графов также имеются эффективные алгоритмы, но они сложнее алгоритма Беллмана-Форда.

Упражнения

4.4.1. Правда ли, что добавление константы к весу каждого ребра не изменяет решение задачи поиска кратчайших путей из одного источника?

4.4.2. Напишите реализацию метода `toString()` для класса `EdgeWeightedDigraph`.

4.4.3. Разработайте реализацию `EdgeWeightedDigraph` для насыщенных графов на основе представления матрицей смежности (двумерный массив весов) (см. упражнение 4.3.9). Параллельные ребра игнорируйте.

4.4.4. Начертите ДКП для источника `0` в орграфе с взвешенными ребрами, полученного удалением вершины `7` из файла `tinyEWD.txt` (см. рис. 4.4.6), и приведите представление этого ДКП родительскими ссылками. Выполните упражнение для того же графа, но с обращенными направлениями всех ребер.

4.4.5. Измените направление ребра `0->2` в файле `tinyEWD.txt` (см. рис. 4.4.6). Приведите два различных ДКП с корнем в вершине `2` для этого измененного орграфа с взвешенными ребрами.

4.4.6. Приведите трассировку, которая демонстрирует процесс вычисления ДКП в орграфе, определенном в упржнении 4.4.5, с помощью "энергичного" варианта алгоритма Дейкстры.

4.4.7. Разработайте вариант класса `DijkstraSP` для поддержки клиентского метода, который возвращает ***второй*** кратчайший путь от `s` до `t` в орграфе с взвешенными ребрами (и возвращает `null`, если кратчайший путь от `s` до `t` единственный).

4.4.8. *Диаметр* орграфа — это длина максимального кратчайшего пути, соединяющего две вершины этого орграфа. Напишите клиент класс `DijkstraSP`, который находит диаметр заданного орграфа `EdgeWeightedDigraph` с неотрицательными весами.

4.4.9. В табл. 4.4.3, взятой из старой дорожной карты, приведены длины кратчайших маршрутов, соединяющих города. Она содержит ошибку. Исправьте ошибку и приведите таблицу, в которой показано, как получить кратчайшие маршруты.

Таблица 4.4.3. Таблица к упражнению 4.4.9

	Провиденс	Уэстерли	Нью-Лондон	Норвич
Провиденс	—	53	54	48
Уэстерли	53	—	18	101
Нью-Лондон	54	18	—	12
Норвич	48	101	12	—

4.4.10. Считайте ребра в орграфе, определенном в упражнении 4.4.4, неориентированными, так что каждое ребро соответствует разнонаправленным ребрам равного веса в орграфе с взвешенными ребрами. Выполните упражнение 4.4.6 для этого соответствующего орграфа.

4.4.11. На основе модели стоимости памяти из раздела 1.4 определите объем памяти, необходимой реализации `EdgeWeightedDigraph` для представления графа с V вершинами и E ребрами.

4.4.12. Адаптируйте классы `DirectedCycle` и `Topological` из раздела 4.2, чтобы в них использовались API `EdgeweightedDigraph` и `DirectedEdge` из данного раздела, и таким образом реализуйте классы `EdgeWeightedCycleFinder` и `EdgeWeightedTopological`.

4.4.13. Покажите в стиле приведенных в тексте трассировок процесс вычисления ДКП с помощью алгоритма Дейкстры а орграфе, полученном удаление ребра `5->7` из графа `tinyEWD.txt` (см. рис. 4.4.6).

4.4.14. Приведите пути, которые будут найдены двумя неверными способами, описанными в подразделах "Заблуждение I" и "Заблуждение II", для примера `tinyEWDn.txt`, который показан там же.

4.4.15. Как отреагирует алгоритм Беллмана-Форда, если на пути из `s` до `v` имеется отрицательный цикл, и выполняется вызов `pathTo(v)`?

4.4.16. Допустим, мы преобразовали класс `EdgeWeightedGraph` в класс `EdgeWeightedDigraph`, создав в этом классе два объекта `DirectedEdge` (по одному в каждом направлении) вместо каждого объекта `Edge` из класса `EdgeWeightedGraph` (как описано для алгоритма Дейкстры в разделе "Вопросы и ответы"), а затем применили алгоритм Беллмана-Форда. Объясните, почему такой способ с треском провалится.

4.4.17. Что случится, если разрешить помещение в очередь нескольких экземпляров вершины в одном и том же проходе алгоритма Беллмана-Форда?

Ответ: время выполнения алгоритма может стать экспоненциальным. Например, опишите, что произойдет при обработке полного взвешенного орграфа, все ребра которого имеют вес –1.

4.4.18. Напишите клиент класса `CPM`, который выводит все критические пути.

4.4.19. Найдите цикл с наименьшим весом (лучшая возможность арбитража) для примера, приведенного в тексте.

4.4.20. Найдите в Интернете или газете таблицу курсов валют. Постройте на ее основе арбитражную таблицу. *Внимание*: не берите таблицы, которые вычислены из нескольких значений и поэтому не содержат информацию, необходимую для реального арбитража.

4.4.21. Покажите в стиле приведенных в тексте трассировок процесс вычисления ДКП с помощью алгоритма Беллмана-Форда для орграфа с взвешенными ребрами из упражнения 4.4.5.

Творческие задачи

4.4.22. *Веса вершин.* Покажите, что вычисление кратчайших путей в орграфах с взвешенными вершинами (неотрицательные веса, вес пути определяется как сумма весов вершин) можно выполнять с помощью построения орграфа с взвешенными ребрами, в котором веса имеют только ребра.

4.4.23. *Кратчайшие пути из источника до стока.* Разработайте API и реализацию на основе варианта алгоритма Дейкстры для решения задачи поиска кратчайшего пути *из источника до стока* в орграфах с взвешенными ребрами.

4.4.24. *Кратчайшие пути с несколькими источниками.* Разработайте API и реализацию на основе алгоритма Дейкстры для решения задачи поиска кратчайших путей *из нескольких источников* в орграфах с положительными весами ребер: для заданного *множества* источников нужно найти лес кратчайших путей, который позволит реализовать метод, возвращающий клиентам кратчайший путь от любого источника до каждой вершины. *Совет*: добавьте к каждому источнику фиктивную вершину с ребром нулевого веса или вначале занесите в очередь с приоритетами все источники, а в соответствующие элементы `distTo[]` занесите нули.

4.4.25. *Кратчайшие пути между двумя подмножествами.* Даны орграф с положительными весами ребер и два различных подмножества вершин S и T. Нужно найти кратчайший путь от любой вершины из S до любой вершины из T. Алгоритм должен выполняться в худшем случае за время, пропорциональное $E \log V$.

4.4.26. *Кратчайшие пути из одного источника в насыщенных графах.* Разработайте вариант алгоритма Дейкстры, который может найти ДКП из заданной вершины в насыщенном орграфе с взвешенными ребрами за время, пропорциональное V^2. Используйте представление матрицей смежности (см. упражнения 4.4.3 и 4.3.29).

4.4.27. *Кратчайшие пути в евклидовых графах.* Измените наши API для ускорения алгоритма Дейкстры, когда известно, что вершинами графа являются точки на плоскости.

4.4.28. *Самые длинные пути в ориентированных ациклических графах.* Разработайте реализацию `AcyclicLP`, которая может решать задачу поиска *самых длинных* путей во взвешенных ориентированных ациклических графах, как описано в утверждении Ф.

4.4.29. *Общая оптимальность.* Завершите доказательство утверждения Ч, показав, что если существует ориентированный путь от `s` до `v` и ни одна вершина на любом пути от `s` до `v` не находится в отрицательном цикле, то существует кратчайший путь от `s` до `v`. (*Совет*: см. утверждение Р.)

4.4.30. *Кратчайшие пути для всех пар вершин в графах с отрицательными циклами.* Сформулируйте API, наподобие реализованного в листинге 4.4.8, для задачи поиска кратчайших путей для всех пар вершин в графах без отрицательных циклов. Разработайте на основе варианта алгоритма Беллмана-Форда реализацию поиска таких весов `pi[v]`, что для любого ребра `v->w` вес ребра плюс разность между `pi[v]` и `pi[w]` — неотрицательная величина. Затем используйте эти веса для переназначения весов в графе, чтобы алгоритм Дейкстры мог эффективно найти все кратчайшие пути в перевзвешенном графе.

4.4.31. *Кратчайший путь для всех пар вершин в линейном графе.* Имеется взвешенный линейный граф — неориентированный связный граф, все вершины которого имеют степени 2, кроме конечных, степень которых равна 1. Предложите алгоритм, который выполняет предобработку графа за линейное время и может возвращать длину кратчайшего пути между любыми двумя вершинами за постоянное время.

4.4.32. *Эвристика проверок родительских вершин.* Измените алгоритм Беллмана-Форда так, чтобы он посещал вершину `v` только в том случае, если ее родительский узел в ДКП `edgeTo[v]` не находится в очереди. Некоторые исследователи отмечают пользу такой эвристики на практике. Докажите, что она корректно вычисляет кратчайшие пути, и что время выполнения такого алгоритма в худшем случае пропорционально EV.

4.4.33. *Кратчайший путь на решетке.* Для заданной матрицы положительных целых чисел размером $N \times N$ нужно найти кратчайший путь от элемента (0, 0) до элемента ($N-1$, $N-1$), если длина пути — это сумма целых чисел на этом пути. Решите эту же задачу, если разрешено двигаться только вправо и вниз.

4.4.34. *Монотонный кратчайший путь.* Для заданного взвешенного орграфа нужно найти *монотонный* кратчайший путь от источника до каждой другой вершины. Путь является монотонным, если он простой (вершины не повторяются), и веса ребер на этом пути либо строго возрастают, либо строго убывают. *Совет*: выполняйте релаксацию ребер в порядке возрастания и найдите лучший путь, затем выполняйте релаксацию в порядке убывания и найдите лучший путь.

4.4.35. *Битонный кратчайший путь.* В заданном орграфе нужно найти *битонный* кратчайший путь от источника до каждой другой вершины (если он существует). Путь является битонным, если на нем существует такая промежуточная вершина, что ребра на пути от источника до этой вершины строго возрастают, а от ребра на пути от этой вершины до конечной строго убывают. Путь должен быть простым (вершины не повторяются).

4.4.36. *Соседи.* Разработайте клиент класса `SP`, который находит все вершины в пределах заданного расстояния от заданной вершины в заданном орграфе с взвешенными ребрами. Время выполнения метода должно быть пропорционально размеру подграфа, индуцированного этими вершинами и инцидентными им, или V (при инициализации структур данных) — в зависимости от того, что больше.

4.4.37. *Критические ребра.* Разработайте алгоритм поиска ребра, удаление которого приводит к максимальному увеличению длины кратчайшего пути из одной заданной вершины в другую заданную вершину в заданном орграфе с взвешенными ребрами.

4.4.38. *Чувствительность.* Разработайте клиент класса `SP`, который анализирует чувствительность ребер взвешенного орграфа относительно заданной пары вершин `s` и `t`. Для этого нужно вычислить логическую матрицу размером $V \times V$, такую, что для каждой пары вершин `v` и `w` элемент в строке `v` и столбце `w` равен `true`, если `v->w` — ребро в орграфе с взвешенными ребрами, вес которого можно увеличить, не увеличив при этом длин кратчайшего пути от `v` до `w`, и `false` в противном случае.

4.4.39. *"Ленивая" реализация алгоритма Дейкстры.* Разработайте реализацию "ленивого" варианта алгоритма Дейкстры, который описан в тексте.

4.4.40. *Критическое ДКП.* Покажите, что МОД в неориентированном графе эквивалентно критическому ДКП в графе: для каждой пары вершин `v` до `w` оно дает соединяющий их путь, в котором самое длинное ребро как можно более короткое.

4.4.41. *Двунаправленный поиск.* Разработайте класс для решения задачи поиска кратчайших путей из источника в сток, который основан на коде наподобие алгоритма 4.9, но вначале в очередь заносится и источник, и сток. При этом ДКП растет из обеих вершин, и основная задача состоит в организации действий после встречи этих частей.

4.4.42. *Худший случай (Дейкстра).* Опишите семейство графов с V вершинами и E ребрами, для которых достигается худший случай времени выполнения для алгоритма Дейкстры.

4.4.43. *Обнаружение отрицательных циклов.* Допустим, что в алгоритм 4.11 добавлен конструктор, который отличается от "родного" только тем, что в нем нет второго аргумента, и вначале обнуляются все элементы `distTo[]`. Покажите, что если клиент использует такой конструктор, то вызов из клиента метода `hasNegativeCycle()` возвратит `true` тогда и только тогда, когда граф содержит отрицательный цикл (а вызов `negativeCycle()` возвратит этот цикл).

Ответ: рассмотрите орграф, сформированный из исходного добавлением нового источника и ребер нулевого веса из этого источника до всех остальных вершин. После первого прохода все элементы `distTo[]` будут равны нулю, и обна-

ружение отрицательного цикла, достижимого из этого источника, эквивалентно обнаружению отрицательного цикла в любом месте исходного графа.

4.4.44. *Худший случай (Беллман-Форд).* Опишите семейство графов с V вершинами и E ребрами, для которых алгоритм 4.11 выполняется за время, пропорциональное VE.

4.4.45. *Быстрый алгоритм Беллмана-Форда.* Разработайте алгоритм, который выходит за рамки линейно-логарифмического времени выполнения для задачи поиска кратчайших путей из одного источника во взвешенных орграфах общего вида для специального случая, когда веса ребер являются целыми числами, ограниченными по абсолютной величине некоторой константой.

4.4.46. *Анимация.* Напишите клиентскую программу, которая выполняет динамические графические анимации алгоритма Дейкстры.

Эксперименты

4.4.47. *Случайные разреженные орграфы с взвешенными ребрами.* Измените решение упражнения 4.3.34, чтобы каждому ребру присваивалось случайное направление.

4.4.48. *Случайные евклидовы орграфы с взвешенными ребрами.* Измените решение упражнения 4.3.35, чтобы каждому ребру присваивалось случайное направление.

4.4.49. *Случайные орграфы с взвешенными ребрами на решетке.* Измените решение упражнения 4.3.36, чтобы каждому ребру присваивалось случайное направление.

4.4.50. *Отрицательные веса I.* Измените генератор случайных орграфов с взвешенными ребрами так, чтобы ребрам присваивались веса в пределах от x до y (и x, и y принадлежат отрезку $[-1, 1]$) путем изменения масштаба и сдвига.

4.4.51. *Отрицательные веса II.* Измените генератор случайных орграфов с взвешенными ребрами так, чтобы ребрам присваивались отрицательные веса — для этого у фиксированного процента (задается клиентом) ребер меняется знак их весов.

4.4.52. *Отрицательные веса III.* Разработайте клиентские программы, которые используют орграфы с взвешенными ребрами, чтобы генерировать другие орграфы, которые содержат значительную часть отрицательных весов, но имеют не более чем несколько отрицательных циклов, для как можно большего диапазона значений V и E.

Тестирование всех алгоритмов со всеми возможными параметрами на всех моделях графов выполнить нереально. Для каждой из приведенных ниже задач напишите клиент, решающий эту задачу, а затем выберите один из описанных выше генераторов, чтобы выполнять эксперименты для данной модели графов. Планируйте эксперименты обдуманно, возможно, на основе результатов предыдущих экспериментов. Напишите краткий анализ полученных результатов и выводы из этих результатов.

4.4.53. *Прогноз.* Оцените, с точностью до порядка 10, размер наибольшего графа с $E = 10V$ ребрами, с которым смогут справиться за 10 секунд ваш компьютер и система программирования, если использовать алгоритм Дейкстры для вычисления всех его кратчайших путей.

4.4.54. *Цена лени.* Эмпирически сравните производительность “ленивого” варианта алгоритма Дейкстры с “энергичным” вариантом, для различных моделей орграфов с взвешенными ребрами.

4.4.55. *Алгоритм Джонсона.* Разработайте реализацию очереди с приоритетами, в которой применяется d-путевая пирамида. Найдите лучшее значение d для различных моделей орграфов с взвешенными ребрами.

4.4.56. *Модель арбитража.* Разработайте модель для генерирования случайных задач арбитража. Цель — получать таблицы, которые максимально похожи на таблицы, использованные в упражнении 4.4.20.

4.4.57. *Модель планирования параллельных работ с предельными сроками.* Разработайте модель для генерирования случайных задач планирования параллельных работ с предельными сроками. Цель — получать нетривиальные задачи, которые, скорее всего, разрешимы.

Глава 5

Строки

Мы общаемся, обмениваясь строками символов. Поэтому многие важные и хорошо знакомые приложения основаны на обработке строк. В данной главе мы рассмотрим классические алгоритмы для работы с (возможно, сложными) задачами наподобие следующих.

- **Обработка информации.** При поиске в Интернете веб-страниц, содержащих указанное ключевое слово, используется приложение обработки строк. В современном мире практически *вся* информация закодирована в виде последовательности строк, и приложения обработки этой информации — это крайне важные приложения обработки строк.
- **Геномика.** Специалисты в области вычислительной биологии работают с *генетическим кодом*, который сводит ДНК к (очень длинным) строкам, образованным из четырех символов — `A`, `C`, `T` и `G`. В последние годы разработаны обширные базы данных, содержащие коды, которые описывают все виды живых организмов, поэтому обработка строк лежит в основе современных исследований в вычислительной биологии.
- **Системы коммуникации.** Посылая текстовое сообщение или электронное письмо, либо загружая электронную книгу, вы пересылаете строку из одного места в другое. Приложения, обрабатывающие строки для этой цели, были первоначальной причиной для разработки алгоритмов обработки строк.
- **Системы программирования.** Программы представляют собой строки. Компиляторы, интерпретаторы и другие приложения, которые преобразуют программы в машинные инструкции — важные приложения, в которых применяются замысловатые приемы. Вообще-то все письменные языки выражаются в виде строк, а еще одной причиной разработки алгоритмов обработки строк была теория формальных языков, т.е. изучение описания множеств строк.

Этот список нескольких важных примеров демонстрирует разнообразие и важность алгоритмов обработки строк.

Вот план данной главы. После знакомства с основными свойствами строк мы еще раз просмотрим в разделах 5.1 и 5.2 API сортировки и поиска из глав 2 и 3. Алгоритмы, задействующие особые свойства строковых ключей, быстрее и гибче алгоритмов, которые были рассмотрены ранее. В разделе 5.3 мы рассмотрим алгоритмы *поиска подстрок* — в том числе знаменитый алгоритм, разработанный Кнутом, Моррисом и Праттом. В разделе 5.4 мы познакомимся с *регулярными выражениями* — основой задачи *сопоставления с шаблоном* (обобщение поиска подстрок) и важным средством поиска *grep*. Эти классические алгоритмы основываются на особых концептуальных устройствах, которые называются *формальные языки* и *конечные автоматы*. Раздел 5.5 посвящен центральному приложению — *сжатию данных*, где размер строки уменьшается до максимально возможного значения.

Правила игры

Для ясности и эффективности наши реализации выражаются в терминах Java-класса `String`, однако мы специально будем использовать минимальное количество операций из этого класса, чтобы облегчить адаптацию наших алгоритмов для обработки других похожих на строки типов данных и для переноса на другие языки программирования.

Мы подробно рассмотрели строки в разделе 1.2, но сейчас еще раз кратко повторим наиболее важные их характеристики.

- **Символы.** Объект `String` представляет собой последовательность символов. Символы имеют тип `char` и могут принимать одно из 2^{16} возможных значений. На протяжении многих десятилетий программисты ограничивались 7- или 8-битными символами ASCII, но во многих современных приложениях требуются 16-битные символы Unicode.
- **Неизменность.** Объекты `String` неизменны, и поэтому их можно использовать в операторах присваивания, в качестве аргументов и при возврате значений из методов, не беспокоясь о том, что их значения могут измениться.
- **Индексация.** Операция, которую мы будем выполнять чаще всего — это *извлечение из строки указанного символа*, и эту операцию выполняет метод `charAt()` из Java-класса `String`. Мы считаем, что вызов `charAt()` выполняет свою работу за *константное* время, как будто строка хранится в массиве `char[]`. Как было сказано в главе 1, это ожидание вполне обоснованно.
- **Длина.** В Java операция *нахождения длины строки* реализована в методе `length()` класса `String`. Мы ожидаем, что этот метод тоже выполняет свою работу за *константное* время, и это ожидание также обоснованно, хотя в некоторых средах программирования требуется некоторая осторожность.
- **Подстроки.** Java-метод `substring()` выполняет операцию *выборки указанной подстроки*. Мы ожидаем, что и этот метод отрабатывает за *константное* время, как в стандартной реализации Java. *Если вы не знакомы с методом* `substring()` *и причиной его работы за константное время, обязательно еще раз прочтите наше обсуждение стандартной реализации строк* в Java в разделе 1.2.
- **Конкатенация.** В Java операция *создания новой строки с помощью приписывания одной строки к другой* является встроенной операцией (используется знак +), которая выполняется за время, пропорциональное длине результата. Например, мы постараемся не формировать строки добавлением по одному символу, т.к. в Java это *квадратичный* процесс. (Для таких вещей в Java имеется класс `StringBuilder`.)
- **Массивы символов.** Строки в Java — определенно не примитивный тип. Стандартная реализация содержит вышеописанные операции для обеспечения программирования в клиентском коде. Однако многие алгоритмы, которые мы рассмотрим, могут работать с низкоуровневым представлением, таким как массив значений `char`, и многим клиентам удобнее работать с подобным представлением, т.к. оно требует меньше и памяти, и времени. Для нескольких алгоритмов стоимость преобразования из одного представления в другое превышает стоимость выполнения алгоритма. Как показано в табл. 5.0.1, различия в коде обработки двух таких представлений минимальны (метод `substring()` реализуется сложнее и поэтому опущен), так что использование одного или другого представления не помешает понимать сам алгоритм.

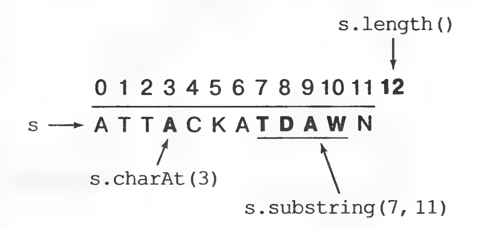


Рис. 5.0.1. *Фундаментальные операции над типом* `String` *с константным временем выполнения*

Таблица 5.0.1. Два способа представления строк в Java (см. рис. 5.0.1)

Операция	Массив символов	Java-строка
Объявление	`char[] a`	`String s`
Обращение к символу по индексу	`a[i]`	`s.charAt(i)`
Длина строки	`a.length`	`s.length()`
Преобразование	`a = s.toCharArray();`	`s = new String(a);`

Понимание эффективности этих операций крайне важно для понимания эффективности нескольких алгоритмов обработки строк. Не во всех языках программирования имеются встроенные реализации строковых типов с такими характеристиками производительности. Например, в широко распространенном языке С операция подстроки и определение длины строки требует времени, пропорционального количеству символов в строке. Адаптация описанных здесь алгоритмов возможна и для таких языков (с помощью реализации API наподобие `String` в Java), но сложности и возможности могут быть другими.

В тексте мы будем использовать в основном тип данных `String`, свободно применяя индексацию и вычисление длины, и реже — извлечение подстроки и конкатенацию. Когда уместно, на сайте книги будет приведен соответствующий код для массивов символов. В приложениях, для которых важна производительность, основным критерием выбора одного из двух представлений часто выступает стоимость доступа к отдельному символу (в типичных реализациях Java `a[i]` выполняется значительно быстрее, чем `s.charAt(i)`).

Алфавиты

Некоторые приложения работают со строками из ограниченного алфавита. В таких случаях зачастую имеет смысл использовать класс `Alphabet`, API которого приведен на рис. 5.0.2.

public class **Alphabet**

	`Alphabet(String s)`	*создание нового алфавита из символов строки* `s`
`char`	`toChar(int index)`	*преобразование индекса в соответствующий символ алфавита*
`int`	`toIndex(char c)`	*преобразование* `c` *в индекс от* `0` *до* `R-1`
`boolean`	`contains(char c)`	*присутствует ли* `c` *в алфавите?*
`int`	`R()`	*основание (количество символов в алфавите)*
`int`	`lgR()`	*количество битов для представления индекса*
`int[]`	`toIndices(String s)`	*преобразование* `s` *в целое число по основанию* `R`
`String`	`toChars(int[] indices)`	*преобразование* `R`*-ичного целого числа в строку этого алфавита*

Рис. 5.0.2. API алфавита

Этот API основан на конструкторе, который принимает в качестве аргумента строку с R символами, задающими алфавит, и методах `toChar()` и `toIndex()` для преобразования (за константное время) символов строки в целые числа от 0 до $R-1$ и наоборот. В нем также имеется метод `contains()` для проверки, присутствует ли в алфавите указный символ, методы `R()` и `lgR()` для определения количества символов в алфавите и количества битов для их представления, а также методы `toIndices()` и `toChars()` для преобразования строк символов алфавита в массивы `int` и наоборот. Для удобства в табл. 5.0.2 мы привели встроенные алфавиты, которые доступны в коде, вроде `Alphabet.UNICODE`. Реализация типа `Alphabet` — несложное упражнение (см. упражнение 5.1.12). Пример клиента приведен в листинге 5.0.1.

Таблица 5.0.2. Стандартные алфавиты

Название	R()	lgR()	Символы
BINARY	2	1	01
DNA	4	2	ACTG
OCTAL	8	3	01234567
DECIMAL	10	4	0123456789
HEXADECIMAL	16	4	0123456789ABCDEF
PROTEIN	20	5	ACDEFGHIKLMNPQRSTVWY
LOWERCASE	26	5	abcdefghijklmnopqrstuvwxyz
UPPERCASE	26	5	ABCDEFGHIJKLMNOPQRSTUVWXYZ
BASE64	6	6	ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghij klmnopqrstuvwxyz0123456789+/
ASCII	128	7	Символы ASCII
EXTENDED_ASCII	256	8	Расширенные символы ASCII
UNICODE16	65536	16	Символы Unicode

Листинг 5.0.1. Типичный клиент класса `Alphabet`

```
public class Count
{
   public static void main(String[] args)
   {
      Alphabet alpha = new Alphabet(args[0]);
      int R = alpha.R();
      int[] count = new int[R];

      String s = StdIn.readAll();
      int N = s.length();
      for (int i = 0; i < N; i++)
         if (alpha.contains(s.charAt(i)))
            count[alpha.toIndex(s.charAt(i))]++;

      for (int c = 0; c < R; c++)
         StdOut.println(alpha.toChar(c)
                        + " " + count[c]);
```

```
% more abra.txt
ABRACADABRA!
% java Count ABCDR < abra.txt
A 5
B 2
C 1
D 1
R 2
```

```
% more pi.txt
3141592653
5897932384
6264338327
9502884197
... [100000 цифр числа пи]
% java Count 0123456789 < pi.txt
0 9999
1 10137
2 9908
3 10026
4 9971
5 10026
6 10028
7 10025
8 9978
9 9902
```

Массивы, индексированные символами

Одна из основных причин использования типа `Alphabet` состоит в том, что многие алгоритмы работают эффективнее при использовании индексированных символами массивов, где с каждым символом связана информация, которую можно выбрать одним обращением к массиву. Работая с Java-типом `String`, нам придется использовать массив размером 65 536; но для работы с типом `Alphabet` нужен массив с одним элементом для каждого символа этого алфавита. Некоторые рассматриваемые нами алгоритмы могут порождать огромное количество таких массивов, и в таких случаях массивы размером 65 536 совершенно неприемлемы. В качестве примера ознакомьтесь с классом `Count` в листинге 5.0.1, который принимает из командной строки строку символов и выводит таблицу счетчиков символов, поданных на стандартный ввод. Массив `count[]`, содержащий счетчики, и является примером индексированного символами массива. Такое вычисление выглядит слишком уж простым, однако на нем основано семейство быстрых сортирующих методов, которые будут рассмотрены в разделе 5.1.

Числа

Как видно из нескольких примеров стандартных алфавитов, мы часто представляем числа в виде строк. Метод `toIndices()` преобразует любую строку из заданного алфа-

вита в число по основанию R, которое содержится в массиве `int[]` со значениями из диапазона от 0 до $R-1$. В некоторых ситуациях первоначальное выполнение такого преобразования приводит к компактному коду, т.к. каждую цифру можно использовать в качестве индекса в индексированном символами массиве. К примеру, если известно, что входные данные содержат лишь символы из алфавита, внутренний цикл в программе `Count` можно заменить более компактным кодом:

```
int[] a = alpha.toIndices(s);
  for (int i = 0; i < N; i++)
    count[a[i]]++;
```

В таком контексте R называется ***основанием*** системы счисления. Несколько алгоритмов, которые мы рассмотрим, часто называются "поразрядными" методами, т.к. они обрабатывают цифру за цифрой.

Несмотря на преимущества использования типов данных вроде `Alphabet` в алгоритмах обработки строк (особенно для небольших алфавитов), в данной книге мы не будем разрабатывать реализации для строк, взятых из произвольного алфавита, потому что:

- в клиентах в основном используется просто тип `String`;
- преобразование в индексы и обратно часто встречается во внутренних циклах и поэтому существенно замедляет реализации;
- код усложняется и поэтому более труден для понимания.

Поэтому мы используем тип `String`, $R = 256$ в коде и R как произвольный параметр при анализе, а там, где уместно, можем обсудить и производительность для обобщенных алфавитов. Полные реализации на основе API `Alphabet` приведены на сайте книги.

5.1. Сортировка строк

Во многих приложениях сортировки определяющие порядок ключи являются строками. В данном разделе мы познакомимся с методами, которые используют особые свойства строк при сортировке строковых ключей и поэтому более эффективны, чем общие методы сортировки, рассмотренные в главе 2.

Мы рассмотрим два фундаментальных способа сортировки строк. Оба они проверены многолетней работой у многих программистов.

Первый способ просматривает символы в ключах справа налево. Такие методы обычно называются сортировкой по младшим разрядам (least-significant-digit — LSD). Слово *разряд* вместо *символ* начали использовать при применении этого же способа к числам различных типов. Если рассматривать строку как число по основанию 256, то просмотр символов справа налево соответствует просмотру чисел, начиная с младших разрядов. Этот способ рекомендуется для сортировки строк, когда все ключи имеют одинаковую длину.

Второй способ просматривает символы в ключах слева направо, начиная с наиболее значащего символа. Его обычно называют сортировкой по старшим разрядам (most-significant-digit — MSD), и в данном разделе мы рассмотрим два таких метода. MSD-сортировки привлекательны тем, что они могут выполнить работу, не обязательно просматривая все символы ключей. Они похожи на быструю сортировку, т.к. разбивают сортируемый массив на независимые части, в которых можно завершить сортировку, рекурсивно применяя к ним тот же метод. Разница в том, что для выполнения разбиения MSD-сортировка строк использует лишь первый символ ключа сортировки, а быстрая сортировка может просматривать весь ключ. Первый способ, который мы рассмотрим, создает раздел для каждого значения ключа, а второй всегда создает три раздела: для ключей, первый символ которых меньше, равен или больше первого символа ключа разбиения.

При анализе сортировки строк важно количество символов в алфавите. В основном мы будем изучать строки из расширенных ASCII-символов (R = 256), однако рассмотрим и строки с символами из гораздо меньших алфавитов (наподобие последовательностей геномов) и гораздо бóльших (например, 65536-символьный алфавит Unicode — международный стандарт для кодирования естественных языков).

Распределяющий подсчет

Для разминки мы рассмотрим простой метод, который эффективен для сортировки небольших целых чисел. Этот метод — *распределяющий подсчет* — полезен и сам по себе, и как основа для тех сортировок строк, которые будут представлены в этом разделе.

Рассмотрим задачу обработки данных, которая может встретиться учителю, выставляющему оценки в классе, где ученики распределены по секциям с номерами 1, 2, 3 и т.д. — см. рис. 5.1.1. В некоторых случаях удобно иметь список класса в разрезе секций. Поскольку номера секций представляют собой небольшие целые числа, здесь как раз годится распределяющий подсчет. Для описания метода мы предположим, что информация хранится в массиве `a[]` элементов, каждый из которых содержит имя и номер раздела, что номера секций являются целыми числами от `0` до `R-1`, и что код `a[i].key()` возвращает код секции для указанного студента. Этот метод выполняется в четыре шага, которые мы сейчас опишем.

Вычисление счетчиков повторений

На первом шаге необходимо вычислить количество повторений каждого значения ключа, используя целочисленный массив `count[]`. Ключи используются для доступа к соответствующим элементам этого массива и увеличения их значений. Если значение ключа равно `r`, мы увеличиваем `count[r+1]`. (Почему +1 — это станет понятным при описании следующего шага.) В примере на рис. 5.1.2 сначала увеличивается значение `count[3]`, т.к. ученик `Anderson` состоит в секции 2, потом два раза увеличивается значение `count[3]`, т.к. ученики `Brown` и `Davis` состоят в третьей секции, и т.д. Учтите, что значение `count[0]` всегда равно нулю, т.к. ни один ученик не приписан к нулевой секции.

Преобразование счетчиков в индексы

Потом накопленные счетчики `count[]` для каждого значения ключа используются для вычисления начальной позиции индекса этого ключа среди упорядоченных элементов. В нашем примере имеются три элемента с ключом 1 и пять элементов с ключом 2, поэтому элементы с ключом 3 начинаются с позиции 8 в упорядоченном массиве. В общем случае для получения начального индекса для элементов с заданным значением ключа нужно сложить счетчики повторений всех меньших значений.

Входные данные: Имя	Секция	Упорядоченный результат (по секциям)	
Anderson	2	Harris	1
Brown	3	Martin	1
Davis	3	Moore	1
Garcia	4	Anderson	2
Harris	1	Martinez	2
Jackson	3	Miller	2
Johnson	4	Robinson	2
Jones	3	White	2
Martin	1	Brown	3
Martinez	2	Davis	3
Miller	2	Jackson	3
Moore	1	Jones	3
Robinson	2	Taylor	3
Smith	4	Williams	3
Taylor	3	Garcia	4
Thomas	4	Johnson	4
Thompson	4	Smith	4
White	2	Thomas	4
Williams	3	Thompson	4
Wilson	4	Wilson	4

ключи — небольшие целые числа

Рис. 5.1.1. *Типичный кандидат для распределяющего подсчета*

```
for (i = 0; i < N; i++)
   count[a[i].key() + 1]++;
```

		count[] 0	1	2	3	4	5
	всегда 0	0	0	0	0	0	0
Anderson	2	0	0	0	**1**	0	0
Brown	3	0	0	0	1	**1**	0
Davis	3	0	0	0	1	**2**	0
Garcia	4	0	0	0	1	2	**1**
Harris	1	0	0	**1**	1	2	1
Jackson	3	0	0	1	1	**3**	1
Johnson	4	0	0	1	1	3	**2**
Jones	3	0	0	1	1	**4**	2
Martin	1	0	0	**2**	1	4	2
Martinez	2	0	0	2	**2**	4	2
Miller	2	0	0	2	**3**	4	2
Moore	1	0	0	**3**	3	4	2
Robinson	2	0	0	3	**4**	4	2
Smith	4	0	0	3	4	4	**3**
Taylor	3	0	0	3	4	**5**	3
Thomas	4	0	0	3	4	5	**4**
Thompson	4	0	0	3	4	5	**5**
White	2	0	0	3	**5**	5	5
Williams	3	0	0	3	5	**6**	5
Wilson	4	0	0	3	5	6	**6**

количество троек

Рис. 5.1.2. *Вычисление счетчиков повторений*

```
for (int r = 0; r < R; r++)
   count[r+1] += count[r];
```

всегда 0

count[]

r	0	1	2	3	4	5
0	0	0	3	5	6	6
1	0	**0**	3	5	6	6
2	0	0	**3**	5	6	6
3	0	0	3	**8**	6	6
4	0	0	3	8	**14**	6
5	0	0	3	8	14	**20**
	0	0	3	8	14	20

количество ключей, меньших 3 (начальный индекс троек в выходных данных)

Рис. 5.1.3. *Преобразование счетчиков в начальные индексы*

Для каждого значения ключа r сумма счетчиков повторений для ключей, меньших r+1, равна сумме счетчиков повторений для ключей, меньших r, плюс count[r] (рис. 5.1.3), поэтому несложно перемещаться слева направо, чтобы преобразовать count[] в индексную таблицу, которую можно использовать для сортировки данных.

Распределение данных

После преобразования массива count[] в индексную таблицу выполняется непосредственно сортировка, во время которой элементы перемещаются во вспомогательный массив aux[]. Каждый элемент перемещается в позицию массива aux[], на которую указывает элемент count[], соответствующий его ключу, а затем этот элемент увеличивается, чтобы сохранялось следующее свойство count[]: для каждого значения ключа r значение count[r] представляет собой индекс позиции в aux[], куда следует поместить следующий элемент со значением ключа r (если такой есть). Этот процесс дает упорядоченный результат после одного прохода по данным — см. рис. 5.1.4 и 5.1.5. *Примечание*: в одном из наших приложений важна *устойчивость* этой реализации: элементы с равными ключами собираются вместе, но в том же самом относительном порядке.

```
for (int i = 0; i < N; i++)
   aux[count[a[i].key()]++] = a[i];
```

count[]

i	1	2	3	4
0	0	3	8	14
1	0	4	8	14
2	0	4	**9**	14
3	0	4	**10**	14
4	0	4	10	15
5	1	4	10	15
6	1	4	**11**	15
7	1	4	11	16
8	1	4	**12**	16
9	2	4	12	16
10	2	5	12	16
11	2	6	12	16
12	3	6	12	16
13	3	7	12	16
14	3	7	12	17
15	3	7	**13**	17
16	3	7	13	18
17	3	7	13	19
18	3	8	13	19
19	3	8	**14**	19
	3	8	14	20
	3	8	14	20

a[0]	Anderson	2	Harris	1	aux[0]
a[1]	Brown	3	Martin	1	aux[1]
a[2]	Davis	3	Moore	1	aux[2]
a[3]	Garcia	4	Anderson	2	aux[3]
a[4]	Harris	1	Martinez	2	aux[4]
a[5]	Jackson	3	Miller	2	aux[5]
a[6]	Johnson	4	Robinson	2	aux[6]
a[7]	Jones	3	White	2	aux[7]
a[8]	Martin	1	**Brown**	**3**	aux[8]
a[9]	Martinez	2	**Davis**	**3**	aux[9]
a[10]	Miller	2	**Jackson**	**3**	aux[10]
a[11]	Moore	1	**Jones**	**3**	aux[11]
a[12]	Robinson	2	**Taylor**	**3**	aux[12]
a[13]	Smith	4	**Williams**	**3**	aux[13]
a[14]	Taylor	3	Garcia	4	aux[14]
a[15]	Thomas	4	Johnson	4	aux[15]
a[16]	Thompson	4	Smith	4	aux[16]
a[17]	White	2	Thomas	4	aux[17]
a[18]	Williams	3	Thompson	4	aux[18]
a[19]	Wilson	4	Wilson	4	aux[19]

Рис. 5.1.4. *Распределение данных (выделены записи с ключом 3)*

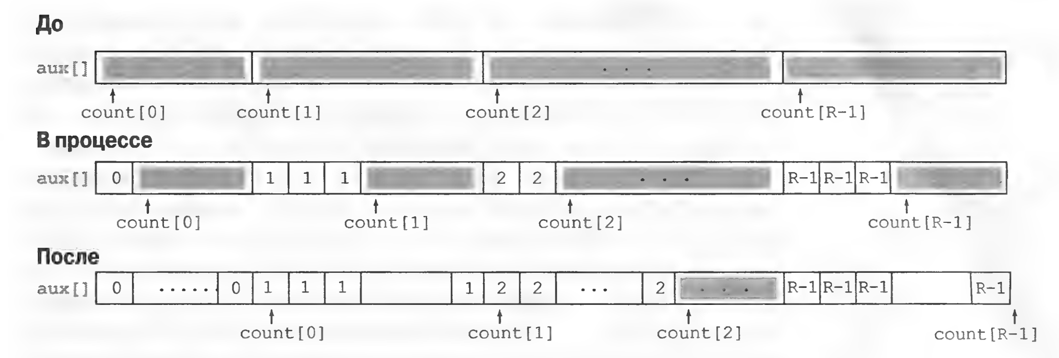


Рис. 5.1.5. Распределяющий подсчет (этап распределения)

Копирование назад

Поскольку сортировка выполнялась с помощью копирования элементов во вспомогательный массив, последним шагом необходимо скопировать полученный результат назад в исходный массив.

Листинг 5.1.1. Распределяющий подсчет (a[].key — целое число из [0, R))

```
int N = a.length;

String[] aux = new String[N];
int[] count = new int[R+1];

// Вычисление счетчиков повторений.
for (int i = 0; i < N; i++)
   count[a[i].key() + 1]++;

// Преобразование счетчиков в индексы.
for (int r = 0; r < R; r++)
   count[r+1] += count[r];

// Распределение записей.
for (int i = 0; i < N; i++)
   aux[count[a[i].key()]++] = a[i];

// Копирование назад.
for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

Утверждение А. Для устойчивой сортировки N элементов, ключи которых представляют собой целые числа от 0 до $R-1$, распределяющему подсчету требуется $8N+3R+1$ обращений к массиву.

Доказательство. Непосредственно следует из кода листинга 5.1.1. Для инициализации массивов нужно $N+R+1$ обращений к массиву. Первый цикл увеличивает на единицу счетчик для каждого из N элементов ($2N$ обращений к массиву), второй выполняет R сложений ($2R$ обращений к массиву), третий выполняет N увеличений счетчиков и N перемещений данных ($3N$ обращений к массиву), и четвертый цикл выполняет N перемещений данных ($2N$ обращений к массиву). Оба этапа перемещений сохраняют относительный порядок одинаковых ключей.

Входные данные	Упорядоченный результат
4PGC938	1ICK750
2IYE230	1ICK750
3CIO720	1OHV845
1ICK750	1OHV845
1OHV845	1OHV845
4JZY524	2IYE230
1ICK750	2RLA629
3CIO720	2RLA629
1OHV845	3ATW723
1OHV845	3CIO720
2RLA629	3CIO720
2RLA629	4JZY524
3ATW723	4PGC938

все строки одинаковой длины

Рис. 5.1.6. *Типичный кандидат для сортировки по младшим цифрам*

Распределяющий подсчет — невероятно эффективный метод для приложений, где ключами являются небольшие целые числа, хотя вспоминают о нем нечасто. Понимание принципа его работы — первый шаг к пониманию сортировки строк. Из утверждения А следует, что распределяющий подсчет преодолевает нижнюю границу $N\log N$, доказанную нами для сортировки. Как же это получается? Дело в том, что утверждение И из раздела 2.2 дает нижнюю границу для количества *сравнений* (когда доступ к данным выполняется только с помощью вызовов `compareTo()`) — а распределяющий подсчет ***не выполняет*** сравнения (обращения к данным с помощью вызовов `key()`). Если R равно N, умноженному на какой-то постоянный коэффициент, то мы имеем сортировку с линейным временем выполнения.

LSD-сортировка строк

Первый метод сортировки строк, который мы рассмотрим, называется *сортировкой по младшим разрядам* (least significant digit first — LSD). Допустим, что автодорожный инженер установил устройство, которое фиксирует номерные знаки всех автомобилей, проезжающих по загруженной автостраде в течение некоторого времени, и хочет узнать количество *различных* транспортных средств, которые пользуются этой трассой. Как было сказано в разделе 2.1, это легко сделать, отсортировав номера, а затем выполнив один проход по списку, подсчитывая различные значения — как в программе `Dedup` (листинг 3.5.1). Автомобильные номера содержат цифры и буквы, поэтому их естественно представлять в виде строк. В простейшем случае (как для номеров автомобилей из Калифорнии — см. рис. 5.1.6) все строки содержат одинаковое количество символов. Такая ситуация часто встречается в сортирующих приложениях: телефонные номера, номера банковских счетов и IP-адреса обычно представляют собой строки фиксированной длины.

Сортировку таких строк можно выполнить с помощью распределяющего подсчета, как показано в алгоритме 5.1 и в примере под ним. Если все строки имеют длину W, то для них W раз выполняется распределяющий подсчет, используя в качестве ключей каждый символ справа налево. Вначале трудно поверить, что этот метод действительно упорядочивает массив — вообще-то он бы и не работал, если бы распределяющий подсчет не был устойчивым. Запомните это и рассматривайте рисунок, когда вы будете читать доказательство корректности работы алгоритма.

Листинг 5.1.2. Алгоритм 5.1. LSD-сортировка строк

```
public class LSD
{
   public static void sort(String[] a, int W)
   {  // Сортировка a[] по старшим W символам.
      int N = a.length;
      int R = 256;
      String[] aux = new String[N];

      for (int d = W-1; d >= 0; d--)
      { // Сортировка с помощью распределяющего подсчета по d-му символу.
```

```
      int[] count = new int[R+1];    // Вычисление счетчиков повторений.
      for (int i = 0; i < N; i++)
         count[a[i].charAt(d) + 1]++;

      for (int r = 0; r < R; r++)    // Преобразование счетчиков в индексы.
        count[r+1] += count[r];

      for (int i = 0; i < N; i++)    // Распределение.
        aux[count[a[i].charAt(d)]++] = a[i];

      for (int i = 0; i < N; i++)    // Копирование назад.
        a[i] = aux[i];
      }
   }
}
```

Чтобы упорядочить массив a[] одинаковой длины W, выполняется W сортировок распределяющим подсчетом для каждой символьной позиции справа налево.

Вначале (W=7)	d = 6	d = 5	d = 4	d = 3	d = 2	d = 1	d = 0	Результат
4PGC938	2IYE230	3CIO720	2IYE230	2RLA629	1ICK750	3ATW723	1ICK750	1ICK750
2IYE230	3CIO720	3CIO720	4JZY524	2RLA629	1ICK750	3CIO720	1ICK750	1ICK750
3CIO720	1ICK750	3ATW723	2RLA629	4PGC938	4PGC938	3CIO720	1OHV845	1OHV845
1ICK750	1ICK750	4JZY524	2RLA629	2IYE230	1OHV845	1ICK750	1OHV845	1OHV845
1OHV845	3CIO720	2RLA629	3CIO720	1ICK750	1OHV845	1ICK750	1OHV845	1OHV845
4JZY524	3ATW723	2RLA629	3CIO720	1ICK750	1OHV845	2IYE230	2IYE230	2IYE230
1ICK750	4JZY524	2IYE230	3ATW723	3CIO720	3CIO720	4JZY524	2RLA629	2RLA629
3CIO720	1OHV845	4PGC938	1ICK750	3CIO720	3CIO720	1OHV845	2RLA629	2RLA629
1OHV845	1OHV845	1OHV845	1ICK750	1OHV845	2RLA629	1OHV845	3ATW723	3ATW723
1OHV845	1OHV845	1OHV845	1OHV845	1OHV845	2RLA629	1OHV845	3CIO720	3CIO720
2RLA629	4PGC938	1OHV845	1OHV845	1OHV845	3ATW723	4PGC938	3CIO720	3CIO720
2RLA629	2RLA629	1ICK750	1OHV845	3ATW723	2IYE230	2RLA629	4JZY524	4JZY524
3ATW723	2RLA629	1ICK750	4PGC938	4JZY524	4JZY524	2RLA629	4PGC938	4PGC938

Утверждение Б. LSD-сортировка устойчиво упорядочивает строки фиксированной длины.

Доказательство. Этот факт полностью зависит от *устойчивости* реализации распределяющего подсчета (см. утверждение А). После (устойчивого) упорядочивания ключей по их i последним символам известно, что любые два ключа будут находиться в массиве в правильном порядке (с учетом только этих символов) либо потому, что первые их i завершающих символов различаются, и тогда сортировка по этому символу расставит их в нужном порядке, либо потому, что первые из их i завершающих символов совпадают, и тогда они остаются упорядоченными в силу устойчивости (и по индукции для i-1).

Другой способ доказать то же самое — заглянуть в будущее: если еще не просмотренные символы в паре ключей одинаковы, то любое различие между этими ключами ограничено уже просмотренными символами, поэтому, в силу устойчивости, ключи уже упорядочены и останутся таковыми. А если не просмотренные символы различны, то просмотренные ранее символы не играют роли, и следующий проход правильно упорядочит пару на основе наиболее значимых различий.

♣J	♦A	♠A
♥6	♥A	♠2
♦A	♣A	♠3
♥A	♠A	♠4
♠K	♠2	♠5
♥J	♣2	♠6
♦Q	♥2	♠7
♣6	♦2	♠8
♠J	♥3	♠9
♣A	♠3	♠10
♦9	♣3	♠J
♥9	♦3	♠Q
♦8	♦4	♠K
♠9	♣4	♥A
♣K	♥4	♥2
♦4	♠4	♥3
♠5	♠5	♥4
♣Q	♦5	♥5
♥3	♣5	♥6
♠2	♥5	♥7
♣10	♥6	♥8
♣9	♣6	♥9
♥7	♠6	♥10
♣4	♦6	♥J
♥4	♥7	♥Q
♦10	♣7	♥K
♠A	♠7	♦A
♦5	♦7	♦2
♠3	♦8	♦3
♥8	♥8	♦4
♣2	♠8	♦5
♦K	♣8	♦6
♠4	♦9	♦7
♣7	♥9	♦8
♥Q	♠9	♦9
♦J	♣9	♦10
♠6	♣10	♦J
♣3	♦10	♦Q
♠7	♠10	♦K
♠8	♥10	♣A
♠10	♣J	♣2
♦3	♥J	♣3
♥10	♠J	♣4
♦7	♦J	♣5
♠Q	♦Q	♣6
♥2	♣Q	♣7
♦2	♥Q	♣8
♣5	♠Q	♣9
♥K	♠K	♣10
♥5	♣K	♣J
♦6	♦K	♣Q
♣8	♥K	♣K

Рис. 5.1.7. *Сортировка колоды карт с помощью LSD-сортировки строк*

Поразрядная LSD-сортировка использовалась старыми машинами сортировки перфокарт, которые были разработаны в начале XX века и были предшественниками компьютеров в обработке коммерческих данных на протяжении нескольких десятков лет. Такие машины могут распределить колоду перфокарт по 10 карманам, в зависимости от отверстий, пробитых в некоторых колонках. Если колода таких карт содержит цифры, пробитые в указанном наборе колонок, оператор может отсортировать карты, пропустив их через машину по правой цифре, затем собрав полученные колоды по порядку и пропустив по второй справа цифре — и т.д. до первой цифры. Физическое складывание колод является устойчивым процессом, который и имитируется сортировкой распределяющим подсчетом. Эта версия поразрядной LSD-сортировки применялась не только для коммерческих приложений (до 1970-х годов), но и многими осторожными программистами (и студентами). Им приходилось хранить свои программы набитыми на перфокарты (по одной строке на каждой карте), в последних колонках которых были пробиты порядковые номера, и если колоду нечаянно рассыпали, ее можно было без труда механически упорядочить по этим номерам.

Этим способом можно упорядочить и колоду обычных игральных карт (рис. 5.1.7): для этого их надо разложить на тринадцать стопок (по одной для каждого вида), сложить их в единую колоду по порядку, а затем разложить на четыре стопки (по одной для каждой масти). Устойчивый процесс раскладывания сохраняет упорядоченность карт в каждой масти, поэтому после складывания стопок в порядке мастей получается отсортированная колода.

Во многих приложениях сортировки строк (а кое-где даже и автомобильных номеров) ключи не обязательно имеют одинаковую длину. LSD-сортировку строк можно адаптировать и для таких ситуаций, но мы оставим эту задачу на самостоятельную проработку, т.к. ниже мы рассмотрим два других метода, которые специально разработаны для ключей переменной длины.

С теоретической точки зрения LSD-сортировка строк важна тем, что в типичных случаях она выполняется за *линейное* время. Каким бы большим ни было значение N, она выполняет W проходов по данным.

Утверждение Б (продолжение). Для сортировки N элементов, ключи которых являются W-символьными строками из R-символьного алфавита, LSD-сортировка строк использует $\sim 7WN + 3WR$ обращений к массиву и объем дополнительной памяти, пропорциональный $N + R$.

Доказательство. Этот метод выполняет W проходов распределяющего подсчета, кроме однократной инициализации массива `aux[]`. Общее значение непосредственно следует из кода и утверждения А.

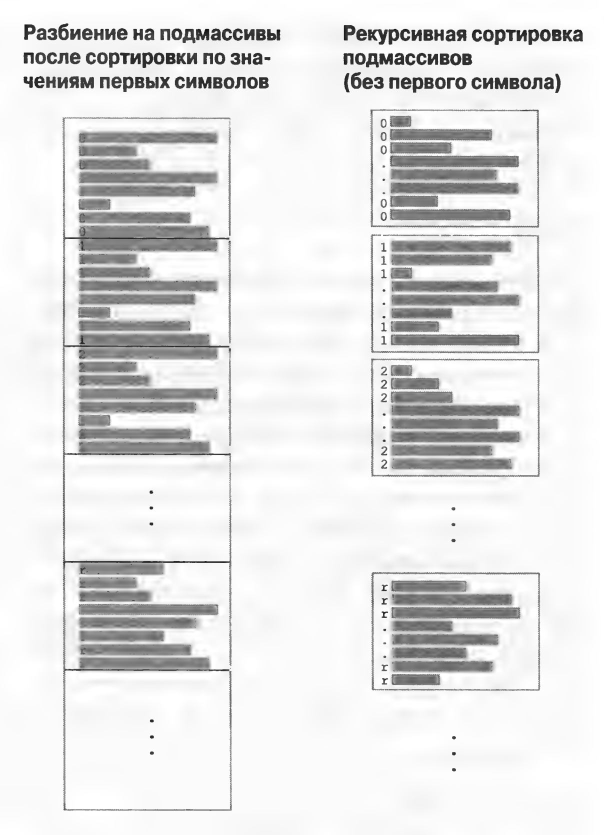


Рис. 5.1.9. Обзор MSD-сортировки строк

Входные данные	Упорядоченный результат
she	are
sells	by
seashells	seashells
by	seashells
the	seashore
seashore	sells
the	sells
shells	she
she	she
sells	shells
are	surely
surely	the
seashells	the

ключи различной длины

Рис. 5.1.10. Типичный кандидат для MSD-сортировки строк

После этих приготовлений реализация MSD-сортировки строк требует совсем немного нового кода — см. алгоритм 5.2. В нем добавлена проверка для перехода на сортировку вставками для небольших подмассивов (это специальная сортировка вставками, мы рассмотрим ее позднее), а в распределяющий подсчет добавлен цикл для выполнения рекурсивных вызовов. Как указано в табл. 5.1.1, значения в массиве `count[]` (сначала это счетчики повторений, потом счетчики для индексации и распределения данных) дают как раз информацию, необходимую для (рекурсивной) сортировки подмассивов, которые соответствуют значению каждого символа.

Листинг 5.1.3. Алгоритм 5.2. MSD-сортировка строк

```
public class MSD
{
  private static int R = 256;          // Основание
  private static final int M = 15;     // Отсечка для небольших подмассивов
  private static String[] aux;         // Вспомогательный массив для распределения

  private static int charAt(String s, int d)
  {  if (d < s.length()) return s.charAt(d); else return -1;  }

  public static void sort(String[] a)
  {
     int N = a.length;
     aux = new String[N];
     sort(a, 0, N-1, 0);
  }

  private static void sort(String[] a, int lo, int hi, int d)
  {  // Сортировка от a[lo] до a[hi], начиная с d-го символа.
     if (hi <= lo + M)
     {  Insertion.sort(a, lo, hi, d); return;  }

     int[] count = new int[R+2];        // Вычисление счетчиков повторений.
     for (int i = lo; i <= hi; i++)
        count[charAt(a[i], d) + 2]++;

     for (int r = 0; r < R+1; r++)      // Преобразование счетчиков в индексы.
        count[r+1] += count[r];

     for (int i = lo; i <= hi; i++)     // Распределение.
        aux[count[charAt(a[i], d) + 1]++] = a[i];

     for (int i = lo; i <= hi; i++)     // Копирование назад.
        a[i] = aux[i - lo];

     // Рекурсивная сортировка для каждого значения символа.
     for (int r = 0; r < R; r++)
        sort(a, lo + count[r], lo + count[r+1] - 1, d+1);
   }
}
```

Чтобы упорядочить массив строк `a[]`, они сортируются по первым символам с помощью распределяющего подсчета, а затем (рекурсивно) сортируются подмассивы, соответствующие каждому значению первого символа.

Таблица 5.1.1. Интерпретация значений `count[]` при MSD-сортировке строк

<table>
<tr><th rowspan="2">По завершении этапа для d-го символа</th><th colspan="5">Значение count[r] равно</th></tr>
<tr><th>r = 0</th><th>r = 1</th><th>r от 2 до R–1</th><th>r = R</th><th>r = R+1</th></tr>
<tr><td>Подсчет повторений</td><td>0 (не используется)</td><td>Количество строк длиной d</td><td colspan="3">Количество строк, значение d-го символа в которых равно r–2</td></tr>
<tr><td>Преобразование счетчиков в индексы</td><td>Начальный индекс в подмассиве для строк длиной d</td><td colspan="3">Начальный индекс в подмассиве строк, значение d-го символа в которых равно r–1</td><td>Не используется</td></tr>
<tr><td rowspan="2">Распределение</td><td colspan="3">Начальный индекс в подмассиве строк, значение d-го символа в которых равно r</td><td colspan="2">Не используется</td></tr>
<tr><td>1 + конечный индекс подмассива для строк длиной d</td><td colspan="3">1 + конечный индекс подмассива строк, значение d-го символа в которых равно r–1</td><td>Не используется</td></tr>
</table>

Заданный алфавит

Стоимость MSD-сортировки строк существенно зависит от количества возможных символов в алфавите. Наш метод сортировки нетрудно изменить так, чтобы он принимал в качестве аргумента объект `Alphabet` и работал эффективнее для клиентов, строки в которых составлены из относительно небольших алфавитов. Для этого нужно:

- сохранить алфавит в переменной экземпляров `alpha` на этапе работы конструктора;
- присвоить в конструкторе переменной `R` значение `alpha.R()`;
- заменить в методе `charAt()` вызов `s.charAt(d)` на `alpha.toIndex(s.charAt(d))`.

В наших примерах используются строки, составленные из строчных букв. LSD-сортировку также нетрудно расширить для учета этой возможности, но, в отличие от MSD-сортировки, обычно это не сильно влияет на производительность.

Код в алгоритме 5.2 на вид прост, но он выполняет довольно замысловатые вычисления. Рекомендуем внимательно изучить высокоуровневую трассировку на рис. 5.1.11 и трассировку рекурсивных вызовов на рис. 5.1.12, чтобы четко уяснить все тонкости работы алгоритма. В этой трассировке используется нулевое пороговое значение `M` отсечки маленьких подмассивов, и можно просмотреть всю сортировку вплоть до самого ее завершения. Строки в данном примере взяты из алфавита `Alphabet.LOWERCASE`, для которого `R = 26`. Учтите, что обычно в приложениях используется алфавит `Alphabet.EXTENDED.ASCII` с `R = 256` или `Alphabet.UNICODE` с `R = 65536`. Для больших алфавитов MSD-сортировка строк проста до опасных пределов: при неумелом использовании она может потребовать огромное количество времени и памяти. Прежде чем перейти к детальному изучению характеристик производительности, мы рассмотрим три важных момента (все они уже упоминались в главе 2), которые вообще-то важны для любого приложения.

Распределяющий подсчет по первому символу | **Рекурсивная сортировка подмассивов**

	Исходный массив			*Подсчет повторений*			*Преобразование счетчиков в индексы*		*Распределение и копирование назад*			*Индексы после завершения этапа распределения*			
0	she	0		0	0		0	0	are	0	0	0	sort(a, 0, 0, 1);	0	are
1	sells	1	a	0	1	a	0	1	by	1	a	1	sort(a, 1, 1, 1);	1	by
2	seashells	2	b	1	2	b	1	2	she	2	b	2	sort(a, 2, 1, 1);	2	sea
3	by	3	c	1	3	c	2	3	sells	3	c	2	sort(a, 2, 1, 1);	3	seashells
4	the	4	d	0	4	d	2	4	seashells	4	d	2	sort(a, 2, 1, 1);	4	seashells
5	sea	5	e	0	5	e	2	5	sea	5	e	2	sort(a, 2, 1, 1);	5	sells
6	shore	6	f	0	6	f	2	6	shore	6	f	2	sort(a, 2, 1, 1);	6	sells
7	the	7	g	0	7	g	2	7	shells	7	g	2	sort(a, 2, 1, 1);	7	she
8	shells	8	h	0	8	h	2	8	she	8	h	2	sort(a, 2, 1, 1);	8	she
9	she	9	i	0	9	i	2	9	sells	9	i	2	sort(a, 2, 1, 1);	9	shells
10	sells	10	j	0	10	j	2	10	surely	10	j	2	sort(a, 2, 1, 1);	10	shore
11	are	11	k	0	11	k	2	11	seashells	11	k	2	sort(a, 2, 1, 1);	11	surely
12	surely	12	l	0	12	l	2	12	the	12	l	2	sort(a, 2, 1, 1);	12	the
13	seashells	13	m	0	13	m	2	13	the	13	m	2	sort(a, 2, 1, 1);	13	the
		14	n	0	14	n	2			14	n	2	sort(a, 2, 1, 1);		
		15	o	0	15	o	2			15	o	2	sort(a, 2, 1, 1);		
		16	p	0	16	p	2			16	p	2	sort(a, 2, 1, 1);		
		17	q	0	17	q	2			17	q	2	sort(a, 2, 1, 1);		
		18	r	0	18	r	2			18	r	2	sort(a, 2, 11, 1);		
		19	s	0	19	s	2			19	s	12	sort(a, 12, 13, 1);		
		20	t	10	20	t	12			20	t	14	sort(a, 14, 13, 1);		
		21	u	2	21	u	14			21	u	14	sort(a, 14, 13, 1);		
		22	v	0	22	v	14			22	v	14	sort(a, 14, 13, 1);		
		23	w	0	23	w	14			23	w	14	sort(a, 14, 13, 1);		
		24	x	0	24	x	14			24	x	14	sort(a, 14, 13, 1);		
		25	y	0	25	y	14			25	y	14	sort(a, 14, 13, 1);		
		26	z	0	26	z	14			26	z	14	sort(a, 14, 13, 1);		
		27		0	27		14			27		14	sort(a, 14, 13, 1);		

начало подмассива s (→ 19 s 2)

1 + конец подмассива s (→ 19 s 12)

Рис. 5.1.11. *Трассировка MSD-сортировки строк: верхний уровень вызова* `sort(a, 0, 14, 0)`

Небольшие подмассивы

Принцип, на котором основана MSD-сортировка строк, довольно эффективен: в типичных ситуациях строки упорядочиваются уже после просмотра лишь нескольких символов в ключах. То есть данный метод быстро разбивает сортируемый массив на небольшие подмассивы. Но это палка о двух концах: необходимо как-то управляться с огромным количеством маленьких подмассивов, т.е. необходим надежный способ эффективной работы с ними. *Небольшие подмассивы сильно влияют на производительность MSD-сортировки строк.* Эта ситуация встречалась нам и для других рекурсивных сортировок (быстрая сортировка и сортировка слиянием), но в случае MSD-сортировки все еще серьезнее. Например, пусть нам понадобилось отсортировать миллионы различных строк символов ASCII ($R = 256$) без отсечки небольших подмассивов. В конце концов, каждая строка окажется оформленной в отдельном подмассиве, поэтому придется сортировать миллионы подмассивов единичного размера. Но для каждой такой сортировки нужно обнулить 258 элементов массива `count[]`, а потом преобразовывать их в индексы. Стоимость этих операций преобладает над остальными операциями сортировки. В случае Unicode ($R = 65536$) сортировка может замедлиться в *тысячи* раз. И действительно, многим ничего не подозревающим клиентам сортировки пришлось наблюдать взрывное увеличение времени выполнения с минут до часов при переходе с ASCII на Unicode, и как раз по этой причине.

Поэтому в случае MSD-сортировки переход на сортировку вставками для небольших подмассивов *необходим*. Чтобы не просматривать заведомо одинаковые символы, мы будем использовать вариант сортировки вставками, приведенный в листинге 5.1.4, в котором имеется дополнительный аргумент `d` и предполагается, что первые `d` символов всех сортируемых строк одинаковы.

Входные данные		d						
she	are	are	are	are	are	are	are	are
sells	by lo	by	by	by	by	by	by	by
seashells	she	sells	seashells	sea	sea	sea	seas	sea
by	sells	seashells	sea	seashells	seashells	seashells	seashells	seashells
the	seashells	sea	seashells	seashells	seashells	seashells	seashells	seashells
sea	sea	sells	sells	sells	sells	sells	sells	sells
shore	shore	seashells	sells	sells	sells	sells	sells	sells
the	shells	she	she	she	she	she	she	she
shells	she	shore	shore	shore	shore	shore	shells	shells
she	sells	shells	shells	shells	shells	shells	shore	shore
sells	surely	she	she	she	she	she	she	she
are	seashells	surely	surely	surely	surely	surely	surely	surely
surely	the hi	the	the	the	the	the	the	the
seashells	the	the	the	the	the	the	the	the

нужно просматривать каждый символ у равных ключей

конец строки находится перед всеми символами

							Выходные данные
are	are	are	are	are	are	are	are
by	by	by	by	by	by	by	by
sea	sea	sea	sea	sea	sea	sea	sea
seashells	seashells	seashells	seashells	seashells	seashells	seashells	seashells
seashells	seashells	seashells	seashells	seashells	seashells	seashells	seashells
sells	sells	sells	sells	sells	sells	sells	sells
sells	sells	sells	sells	sells	sells	sells	sells
she	she	she	she	she	she	she	she
shells	shells	shells	shells	she	she	she	she
she	she	she	she	shells	shells	shells	shells
shore	shore	shore	shore	shore	shore	shore	shore
surely	surely	surely	surely	surely	surely	surely	surely
the	the	the	the	the	the	the	the
the	the	the	the	the	the	the	the

Рис. 5.1.12. Трассировка рекурсивных вызовов MSD-сортировки строк (без отсечки для небольших подмассивов, подмассивы размером 0 и 1 не показаны)

Эффективность этого кода зависит от того, является ли время выполнения операции `substring()` константным. Как и в случае быстрой сортировки и сортировки вставками, основной выигрыш от этой модификации достигается для небольших значений отсечки, но соответствующая экономия более существенна.

На рис. 5.1.13 показан результат эксперимента, где видно, что переход на сортировку вставками для подмассивов размером 10 элементов и менее снижает в типичных ситуациях время выполнения в 10 раз.

Листинг 5.1.4. Сортировка вставками для строк с одинаковыми первыми d символами

```
public static void sort(String[] a, int lo, int hi, int d)
{  // Сортировка от a[lo] до a[hi], начиная с d-го символа.
   for (int i = lo; i <= hi; i++)
      for (int j = i; j > lo && less(a[j], a[j-1], d); j--)
         exch(a, j, j-1);
}

private static boolean less(String v, String w, int d)
{  return v.substring(d).compareTo(w.substring(d)) < 0;  }
```

Равные ключи

Еще один опасный момент в MSD-сортировке строк — она может работать относительно медленно для подмассивов, содержащих большое количество одинаковых ключей. При наличии достаточно большого количества подстрок отсечка для небольших подмассивов не срабатывает, и приходится выполнять рекурсивные вызовы для каждого символа одинаковых ключей. Более того, распределяющий подсчет неэффективен для определения того, что все символы равны: приходится не только просматривать каждый символ и перемещать каждую строку, но и обнулять все счетчики, преобразовывать их в индексы и т.д. Поэтому худший случай для MSD-сортировки строк — когда все ключи равны. Эта же проблема возникает и при наличии длинных общих префиксов в большом количестве ключей, а такая ситуация часто встречается в реальных приложениях.

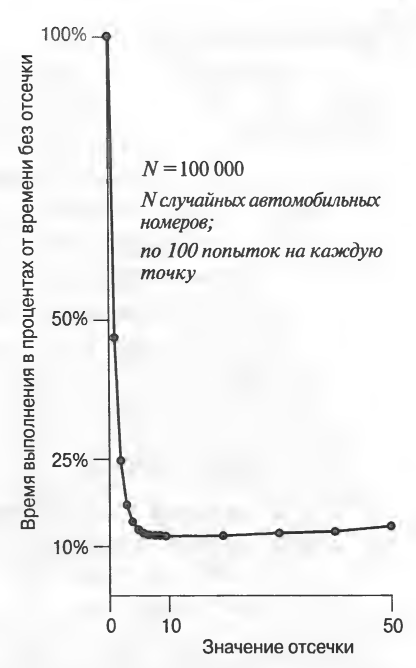


Рис. 5.1.13. Эффект отсечки для небольших подмассивов в MSD-сортировке строк

Дополнительная память

Для выполнения разбиений MSD использует два вспомогательных массива: временный массив для распределения ключей `aux[]` и массив для хранения счетчиков, которые преобразуются в индексы разделов — `count[]`. Массив `aux[]` имеет размер N, и его можно создать вне рекурсивного метода `sort()`. Эту дополнительную память можно сэкономить, пожертвовав устойчивостью (см. упражнение 5.1.17), но в реальных применениях MSD-сортировки строк это не основная забота. А вот память для массива `count[]` может играть важную роль: его невозможно создать за пределами рекурсивного метода `sort()`, как сказано ниже в утверждении Г.

Модель случайных строк

Для изучения производительности MSD-сортировки строк мы используем *модель случайных строк*, где каждая строка состоит из (независимо) случайных символов без ограничения на длину строк. Поэтому длинные равные ключи можно игнорировать, как крайне невероятный случай (рис. 5.1.14). Поведение MSD-сортировки в этой модели аналогично ее поведению в модели со случайными ключами фиксированной длины и в модели для типичных реальных данных. Как мы увидим, во всех трех случаях MSD-сортировка строк обычно просматривает лишь несколько символов в начале ключей.

Случайные (сублинейное время)	Неслучайные с повторениями (почти линейное время)	Худший случай (линейное время)
1EIO402	are	1DNB377
1HYL490	by	1DNB377
1ROZ572	sea	1DNB377
2HXE734	seashells	1DNB377
2IYE230	seashells	1DNB377
2XOR846	sells	1DNB377
3CDB573	sells	1DNB377
3CVP720	she	1DNB377
3IGJ319	she	1DNB377
3KNA382	shells	1DNB377
3TAV879	shore	1DNB377
4CQP781	surely	1DNB377
4QGI284	the	1DNB377
4YHV229	the	1DNB377

Рис. 5.1.14. Символы, просматриваемые MSD-сортировкой строк

Производительность

Время работы MSD-сортировки строк зависит от входных данных. Для методов, основанных на сравнениях, в основном был важен *порядок* ключей, но в данном случае важно другое — *значения* ключей.

- Для *случайных* входных данных MSD-сортировка строк проверяет лишь столько символов, сколько нужно для различения ключей, и время выполнения *сублинейно* относительно количества символов в данных (просматривается только небольшая часть входных символов).
- Для *неслучайных* входных данных MSD-сортировка строк может выполняться за сублинейное время, но ей может потребоваться проверка большего количества символов, чем для случайных данных (в зависимости от данных). В частности, в случае равных ключей может потребоваться просмотр всех символов, поэтому при наличии значительного количества одинаковых ключей время выполнения приближается к линейному относительно количества символов в данных.
- В *худшем случае* MSD-сортировка строк просматривает все символы в ключах, поэтому время выполнения *линейно* относительно количества символов в данных (как и в LSD-сортировке строк). Худший случай достигается тогда, когда все входные строки одинаковы.

В некоторых приложениях используются различные ключи, которые хорошо описываются моделью случайных строк, другие работают со значительным количеством равных ключей или длинных общих префиксов, и тогда время сортировки приближается к худшему случаю. Например, наше приложение сортировки автомобильных номеров может попасть в любую из этих категорий: если инженер возьмет часовые данные с загруженной межобластной автострады, повторений будет очень мало, и вполне применима случайная модель, а если взять данные за неделю для районной дороги, то данные будут содержать многочисленные повторения, и производительность сильно сместится к худшему случаю.

Утверждение В. Для сортировки N случайных строк из R-символьного алфавита MSD-сортировке требуется просмотреть в среднем порядка $N\log_R N$ символов.

Набросок доказательства. Мы ожидаем, что все подмассивы будут иметь примерно одинаковые размеры, поэтому производительность будет приблизительно описываться рекуррентным соотношением $C_N = RC_{N/R} + N$, и обобщение рассуждения из главы 2 приводит к утверждаемому результату. Это описание ситуации не вполне точно, т.к. N/R — не обязательно целое число, и подмассивы имеют одинаковый размер только в среднем (и в силу конечности количества символов в реальных ключах). Эти эффекты не так сильно влияют на MSD-сортировку строк, как стандартная быстрая сортировка, поэтому старший член в выражении для времени выполнения дает решение этого рекуррентного уравнения. Детальный анализ, который доказывает этот факт, является классическим примером анализа алгоритмов, и впервые был выполнен Кнутом (Knuth) в начале 1970-х годов.

В качестве пищи для размышления можно подсказать, почему доказательство этого утверждения выходит за рамки данной книги. Длина ключей не играет никакой роли, поскольку модель случайных строк допускает произвольную длину ключей. Для любого заданного количества символов существует отличная от нуля вероятность, что два ключа совпадут, но эта вероятность настолько мала, что не играет роли в оценках производительности.

Как уже было сказано, количество просмотренных символов — не все, что важно для оценки MSD-сортировки строк. Необходимо также учитывать время и память, необходимые для подсчета повторений и преобразования счетчиков в индексы.

Утверждение Г. Для сортировки N строк из R-символьного алфавита MSD-сортировке нужно от $8N+3R$ до $\sim 7wN+3WR$ обращений к массиву, где w — средняя длина строк.

Доказательство. Непосредственно следует из кода, утверждений А и Б. В лучшем случае MSD-сортировке нужен лишь один проход, а в худшем она работает вроде LSD-сортировки.

В случае малых N преобладает влияние R. Точный анализ полной стоимости труден и сложен, но эффект этой стоимости можно оценить, просто рассмотрев небольшие подмассивы, если все ключи различны. Без отсечки небольших подмассивов каждый ключ окажется в собственном подмассиве, поэтому для этих подмассивов необходимо NR обращений. Если выполнять отсечку для небольших подмассивов размером M, то получится примерно N/M подмассивов размером M, и мы имеем компромисс между NR/M обращениями к массивам и $NM/4$ сравнениями — откуда следует, что нужно выбрать M пропорциональным квадратному корню из R.

Утверждение Г (продолжение). Для сортировки N строк из R-символьного алфавита MSD-сортировка строк требует в худшем случае объем памяти, пропорциональный R, умноженному на длину самой длинной строки (плюс N).

Доказательство. Массив `count[]` должен быть создан в методе `sort()`, поэтому общий объем необходимой памяти пропорционален R, умноженному на глубину рекурсии (плюс N для вспомогательного массива). Точнее, глубина рекурсии равна длине самой длинной строки, которая является префиксом для двух или более сортируемых строк.

Как только что было сказано, одинаковые ключи доводят глубину рекурсии до пропорциональной длине ключей. Непосредственный практический вывод из утверждения Г состоит в том, что MSD-сортировка строк запросто может исчерпать ресурсы времени или памяти при сортировке длинных строк из большого алфавита, особенно при наличии длинных одинаковых ключей. Например, в случае более чем `M` одинаковых 1000-символьных строк из алфавита `Alphabet.UNICODE` методу `MSD.sort()` потребуются более 65 миллионов счетчиков!

Основная трудность при обеспечении максимальной эффективности для MSD-сортировки длинных строк — нехватка случайности в данных. Зачастую они содержат длинные ряды одинаковых ключей, или часть их находится в узком диапазоне. Например, приложение обработки данных о студентах может содержать ключи для года выпуска (4 байта, но одно из четырех различных значений), названия штатов (до 10 байтов, но одно из 50 различных значений) и пол (1 байт с одним из двух возможных значений), а также имя (больше похоже на случайные строки, но, как правило, не очень короткие, с неравномерным распределением букв и хвостовыми пробелами в поле фиксированной длины). Подобные особенности приводят при выполнении MSD-сортировки к большому количеству пустых подмассивов. А теперь мы рассмотрим элегантный способ справиться с такими ситуациями.

Трехчастная быстрая сортировка строк

Быструю сортировку можно адаптировать к виду MSD-сортировки, используя трехчастное разбиение по старшему символу ключей и переходя к следующему символу только в среднем подмассиве (старшие символы в ключах которых равны центральному символу) — см. рис. 5.1.15 и 5.1.16. Этот метод несложно реализовать — см. алгоритм 5.3 в листинге 5.1.5. Для этого нужно лишь добавить в рекурсивный метод из алгоритма 2.5 аргумент, который будет хранить текущий символ, изменить код трехчастного разбиения для использования этого символа и соответственно изменить рекурсивные вызовы.

Листинг 5.1.5. Алгоритм 5.3. Трехчастная быстрая сортировка строк

```
public class Quick3string
{
   private static int charAt(String s, int d)
   {  if (d < s.length()) return s.charAt(d); else return -1;  }

   public static void sort(String[] a)
   {  sort(a, 0, a.length - 1, 0);  }

   private static void sort(String[] a, int lo, int hi, int d)
   {
      if (hi <= lo) return;
      int lt = lo, gt = hi;
      int v = charAt(a[lo], d);
      int i = lo + 1;
      while (i <= gt)
      {
         int t = charAt(a[i], d);
         if      (t < v) exch(a, lt++, i++);
         else if (t > v) exch(a, i, gt--);
         else            i++;
      }
      // a[lo..lt-1] < v = a[lt..gt] < a[gt+1..hi]

      sort(a, lo, lt-1, d);
      if (v >= 0) sort(a, lt, gt, d+1);
      sort(a, gt+1, hi, d);
   }
}
```

Чтобы упорядочить массив строк `a[]`, они разбиваются на три части по первым символам, а затем (рекурсивно) сортируются три полученных подмассива: строки, в которых первый символ меньше центрального символа, строки, в которых первый символ равен центральному символу (и этот первый символ уже исключается из сортировки), и строки, в которых первый символ больше центрального символа.

Трехчастная быстрая сортировка строк выполняет вычисления в другом порядке, но сводится к сортировке массива по старшим символам ключей (с помощью быстрой сортировки), а затем к рекурсивному применению метода к остальным ключам. При сортировке строк этот метод ведет себя лучше как обычной быстрой сортировки, так и MSD-сортировки строк, являясь, по сути, гибридом данных двух алгоритмов.

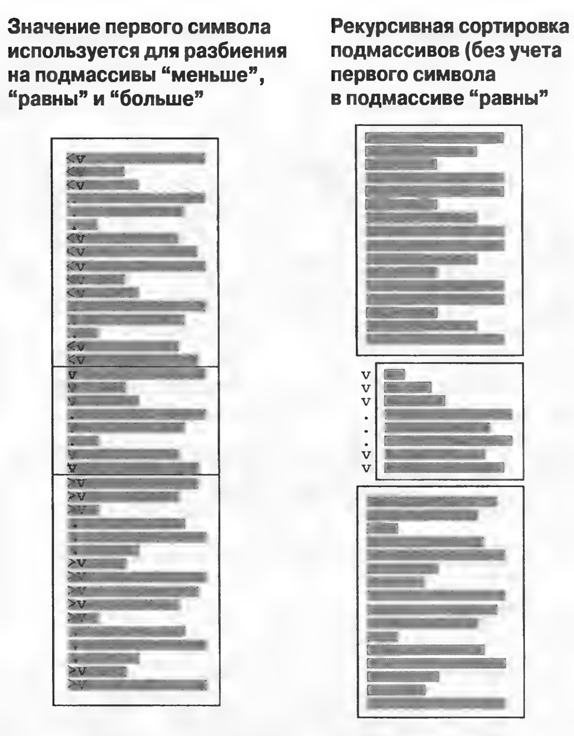


Рис. 5.1.15. Обзор трехчастной быстрой сортировки строк

Трехчастная быстрая сортировка строк разбивает массив только на три части, поэтому в ней выполняется больше перемещений данных, чем в MSD-сортировке строк, если количество непустых разделов велико — ведь ей приходится выполнять несколько трехчастных разбиений, чтобы получить эффект многочастного разбиения. Но MSD-сортировка может создавать большие количества (пустых) подмассивов, а трехчастная быстрая сортировка всегда создает лишь три. Таким образом, трехчастная быстрая сортировка строк хорошо приспособлена для обработки равных ключей, ключей с длинными общими префиксами, ключей из узкого диапазона и небольших массивов — т.е. всех ситуаций, где MSD-сортировка работает медленно. Очень важно то, что разбиение приспосабливается к различным видам структуры в различных частях ключа. Кроме того, как и быстрая сортировка, трехчастная быстрая сортировка строк не использует дополнительную память (кроме неявного стека для поддержки рекурсии), а это серьезное преимущество по сравнению с MSD-сортировкой, которая выделяет память для счетчиков повторений и для вспомогательного массива.

На рис. 5.1.17 показаны все рекурсивные вызовы, которые выполняет программа `Quick3string` для нашего примера. Каждый подмассив сортируется с помощью в точности трех рекурсивных вызовов, за исключением пропуска вызовов при достижении концов (равных) строк в среднем подмассиве.

Как обычно, на практике стоит рассмотреть различные стандартные усовершенствования в реализации алгоритма 5.3.

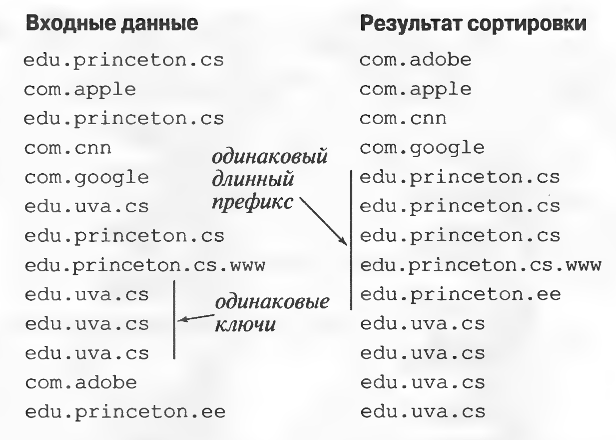


Рис. 5.1.16. *Типичный кандидат для трехчастной быстрой сортировки строк*

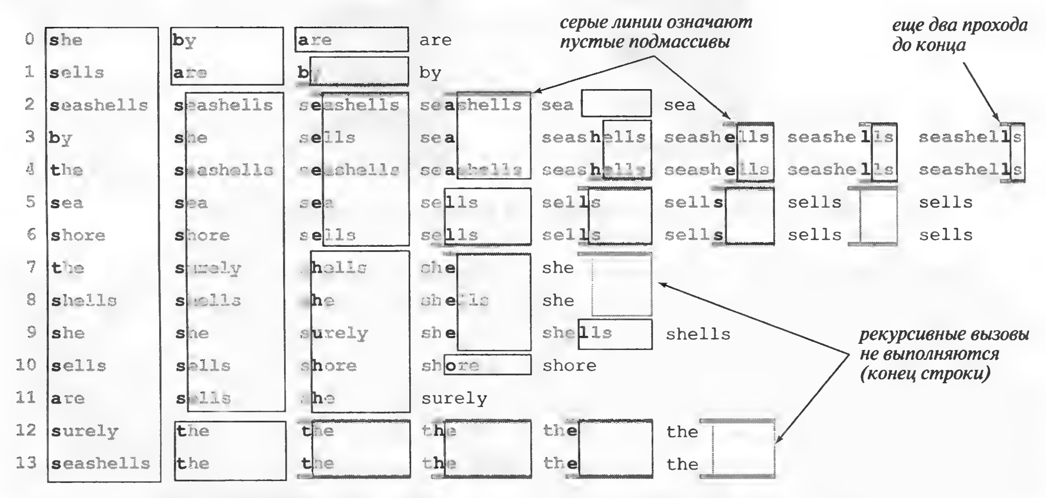


Рис. 5.1.17. *Трассировка рекурсивных вызовов трехчастной быстрой сортировки строк (без отсечки небольших подмассивов)*

Небольшие подмассивы

В любом рекурсивном алгоритме можно повысить эффективность, особым образом обрабатывая небольшие подмассивы. В данном случае мы используем сортировку вставками из листинга 5.1.4, которая пропускает заведомо равные символы. Повышение производительности, которое дает это изменение, обычно существенно, хотя и не настолько, как для MSD-сортировки строк.

Ограниченный алфавит

Для работы со специальными алфавитами в каждый из методов можно добавить аргумент `alpha` типа `Alphabet` и заменить в методе `charAt()` выражение `s.charAt(d)` на `alpha.toIndex(s.charAt(d))`. В данном случае это ничего не дает, и добавление такого кода, скорее всего, замедлит работу алгоритма, т.к. он добавляется во внутренний цикл.

Рандомизация

В любой быстрой сортировке обычно имеет смысл предварительно перетасовать массив или использовать случайный центральный элемент, обменивая первый элемент со случайно выбранным. Это делается в основном для защиты от худшего случая, когда массив уже упорядочен или почти упорядочен.

На самом деле стандартная быстрая сортировка и все другие виды сортировки из главы 2 применительно к строковым ключам являются MSD-сортировкой, т.к. метод `compareTo()` из класса `String` просматривает символы слева направо. То есть метод `compareTo()` обращается только к старшим символам, если они различны, к двум старшим символам, если первые символы совпадают, и т.д. Например, если все первые символы строк различны, то стандартные методы сортировки просмотрят только эти символы, таким образом автоматически получив примерно тот же выигрыш в производительности, который мы пытались получить в MSD-сортировке. Главный принцип, на котором основана трехчастная быстрая сортировка — особые действия в случае равенства старших символов. Ведь алгоритм 5.3 можно рассматривать как стандартную быструю сортировку с отслеживанием заведомо равных ведущих символов. Стандартному алгоритму приходится просматривать все эти символы при каждом сравнении, а трехчастный алгоритм избегает этого.

Производительность

Рассмотрим случай длинных строковых ключей (для простоты — одинаковой длины), у которых большинство старших символов совпадают. В такой ситуации время выполнения стандартной быстрой сортировки пропорционально длине строк, *умноженной* на $2N\ln N$, а время выполнения трехчастной быстрой сортировки пропорционально N, умноженному на длину строк (для обнаружения всех одинаковых старших символов), *плюс* $2N\ln N$ сравнений символов (для выполнения сортировки по оставшимся коротким ключам). То есть трехчастная быстрая сортировка строк требует вплоть до в $2\ln N$ раз меньше сравнений символов, чем обычная быстрая сортировка. В практических приложениях сортировки ключи с подобными характеристиками встречаются часто.

> **Утверждение Д.** Для сортировки массива из N случайных строк трехчастная быстрая сортировка строк выполняет в среднем $\sim 2N\ln N$ сравнений.
>
> **Доказательство.** Этот результат можно получить двумя показательными способами. Если считать метод эквивалентным разбиениям в быстрой сортировке по ведущим символам с последующим (рекурсивным) применением этого же метода для подмассивов, то неудивительно, что общее количество операций примерно совпадает с обычной быстрой сортировкой — но это сравнения отдельных символов, а не полностью ключей. А если считать метод заменой распределяющего подсчета на быструю сортировку, то можно ожидать, что время выполнения $N\log_R N$ из утверждения Г следует умножить на $2\ln R$, т.к. для упорядочения R символов быстрой сортировке требуется выполнить $2R\ln R$ шагов, в отличие от R шагов для тех же символов в MSD-сортировке строк. Полное доказательство мы приводить не будем.

Как уже было сказано выше, рассмотрение модели случайных строк полезно, но для прогноза производительности в практических ситуациях нужен более детальный анализ. Исследователи хорошенько изучили этот алгоритм и доказали, что никакой алгоритм не может превзойти трехчастную быструю сортировку строк (по количеству сравнений символов) больше, чем в постоянное число раз, при очень общих допущениях. Чтобы

оценить гибкость, заметьте, что трехчастная быстрая сортировка не зависит непосредственно от размера алфавита.

Пример: журнал работы в Интернете

В качестве примера несомненного превосходства трехчастной быстрой сортировки строк можно рассмотреть типичную современную задачу обработки данных. Допустим, имеется веб-сайт, и нужно проанализировать генерируемый им веб-трафик. Системный администратор может предоставить журнал, в котором зафиксированы все транзакции этого сайта. Среди информации, связанной с такими транзакциями, присутствуют доменные имена компьютеров, откуда отсылались данные. Пусть, к примеру, на сайте по продажам книг сформирован журнал транзакций за неделю `week.log.txt`. Почему трехчастная быстрая сортировка строк легко справится с таким файлом? Потому что в упорядоченном результате имеется много одинаковых длинных префиксов, которые этот метод не должен просматривать многократно.

Каким алгоритмом сортировки строк воспользоваться?

Конечно, нам интересно, как соотносятся изученные здесь методы сортировки строк с методами общего назначения, которые были рассмотрены в главе 2. В табл. 5.1.2 приведены важные характеристики алгоритмов сортировки строк, которые были описаны в данном разделе (для сравнения добавлены строки для быстрой сортировки, сортировки слиянием и трехчастной быстрой сортировки из главы 2).

Таблица 5.1.2. Характеристики производительности алгоритмов сортировки строк

Алгоритм	Устойчив?	На месте?	Порядок роста типичного количества вызовов `charAt()` для сортировки N строк из R-символьного алфавита (средняя длина w, максимальная длина W)		Сильные стороны
			Время выполнения	Дополнительная память	
Сортировка вставками для строк	да	да	от N до N^2	1	Небольшие массив, упорядоченные массивы
Быстрая сортировка	нет	да	$N\log^2 N$	$\log N$	Общего назначения при нехватке памяти
Сортировка слиянием	да	нет	$N\log^2 N$	N	Устойчивая сортировка общего назначения
Трехчастная быстрая сортировка	нет	да	от N до $N\log N$	$\log N$	Большое количество одинаковых ключей
LSD-сортировка строк	да	нет	NW	N	Короткие строки фиксированной длины
MSD-сортировка строк	да	нет	от N до Nw	$N + WR$	Случайные строки
Трехчастная быстрая сортировка строк	нет	да	от N до Nw	$W + \log N$	Общего назначения, строки с одинаковыми длинными префиксами

Как и в главе 2, умножение этих порядков роста на константы, соответствующие алгоритмам и данным, позволяет неплохо прогнозировать время выполнения.

Как было показано в рассмотренных примерах и будет показано в упражнениях, для разных конкретных ситуаций удобны различные методы с соответствующими значениями параметров. В руках эксперта (возможно, это уже и вы!) в некоторых ситуациях возможен значительный выигрыш.

Вопросы и ответы

Вопрос. **Используется ли один из этих методов в системной сортировке Java для сортировки объектов `String`?**

Ответ. Нет, но стандартная реализация содержит быстрое сравнение строк, которое делает стандартные сортировки сравнимыми с рассмотренными здесь методами.

Вопрос. **Значит, для сортировки ключей типа `String` следует использовать системную сортировку?**

Ответ. В Java — пожалуй, да. Правда, если у вас огромные количества строк или нужно сортировать их очень быстро, то можно перейти со значений `String` на массивы `char` и использовать поразрядную сортировку.

Вопрос. **Откуда взялись множители $\log^2 N$ в табл. 5.1.2?**

Ответ. Они означают, что большинство сравнений в этих алгоритмах выполняется между ключами с общим префиксом длиной $\log N$. Последние исследования на основе тщательного математического анализа подтвердили этот факт для случайных строк (см. сайт книги).

Упражнения

5.1.1. Разработайте реализацию сортировки, которая подсчитывает количество различных значений ключей, а потом использует таблицу символов для сортировки массива с помощью распределяющего подсчета. (Этот метод *не* годится в случае большого количества различных значений ключей.)

5.1.2. Приведите трассировку работы LSD-сортировки для строковых ключей

`no is th ti fo al go pe to co to th ai of th pa`

5.1.3. Приведите трассировку работы MSD-сортировки для строковых ключей

`no is th ti fo al go pe to co to th ai of th pa`

5.1.4. Приведите трассировку работы трехчастной быстрой сортировки для строковых ключей

`no is th ti fo al go pe to co to th ai of th pa`

5.1.5. Приведите трассировку работы MSD-сортировки для строковых ключей

`now is the time for all good people to come to the aid of`

5.1.6. Приведите трассировку работы трехчастной быстрой сортировки для строковых ключей

`now is the time for all good people to come to the aid of`

5.1.7. Разработайте реализацию распределяющего подсчета, в котором используется массив объектов `Queue`.

5.1.8. Приведите количество символов, просматриваемых MSD-сортировкой строк и трехчастной быстрой сортировкой строк для файла из N ключей `a`, `aa`, `aaa`, `aaaa`, `aaaaa`, ...

5.1.9. Разработайте реализацию LSD-сортировки строк, которая способна работать со строками переменной длины.

5.1.10. Чему равно общее количество символов, просматриваемых в худшем случае трехчастной быстрой сортировкой при упорядочении N строк фиксированной длины W?

Творческие задачи

5.1.11. *Сортировка с помощью очередей.* Реализуйте MSD-сортировку строк с помощью очередей. Для каждого кармана используйте отдельную очередь. На первом проходе по сортируемым элементам вставляйте каждый элемент в очередь, соответствующую значению ее старшего символа. Потом отсортируйте подсписки и объедините содержимое очередей в единый результат. Обратите внимание, что в этом методе не требуется хранить массивы `count[]` внутри рекурсивного метода.

5.1.12. *Алфавит.* Разработайте реализацию API `Alphabet`, приведенного на рис. 5.0.2, и воспользуйтесь ей для разработки LSD- и MSD-сортировки строк из произвольных алфавитов.

5.1.13. *Гибридная сортировка.* Обдумайте идею применения стандартной MSD-сортировки строк для больших массивов, чтобы воспользоваться преимуществами многочастного разбиения, и трехчастной быстрой сортировкой для меньших массивов, чтобы избежать появления больших количеств пустых карманов.

5.1.14. *Сортировка массивов.* Разработайте метод, в котором трехчастная быстрая сортировка строк применяется для ключей, представляющих собой *массивы* целых чисел.

5.1.15. *Сублинейная сортировка.* Разработайте реализацию сортировки для целочисленных значений, которая выполняет два прохода по сортируемому массиву: LSD-сортировка по старшим 16 битам ключей, а затем сортировка вставками.

5.1.16. *Сортировка связного списка.* Разработайте реализацию сортировки, которая принимает в качестве аргумента связный список узлов с ключами типа `String` и упорядочивает его (возвращает ссылку на узел с наименьшим ключом). Воспользуйтесь трехчастной быстрой сортировкой строк.

5.1.17. *Распределяющий подсчет на месте.* Разработайте версию распределяющего подсчета, которая использует только постоянный объем дополнительной памяти. Докажите устойчивость этой версии или приведите контрпример.

Эксперименты

5.1.18. *Случайные десятичные ключи.* Напишите статический метод `randomDecimalKeys`, который принимает в качестве аргументов целочисленные значения `N` и `W` и возвращает массив `N` строковых значений, каждое из которых является `W`-значным десятичным числом.

5.1.19. *Случайные автомобильные номера.* Напишите статический метод `randomPlatesCA`, который принимает в качестве аргумента целое значение `N` и возвращает массив из `N` значений типа `String`, которые представляют собой автомобильные номера вроде приведенных в примерах данного раздела.

5.1.20. *Случайные слова фиксированной длины.* Напишите статический метод `randomFixedLengthWords`, который принимает в качестве аргументов целочисленные значения `N` и `W` и возвращает массив `N` строковых значений из `W` символов алфавита.

5.1.21. *Случайные элементы.* Напишите статический метод `randomItems`, который принимает в качестве аргумента целое значение `N` и возвращает массив из `N` строковых значений длиной от 15 до 30 символов, которые состоят из четырех полей: 4-символьное поле с одной из 10 фиксированных строк; 10-символьное поле с одной из 50 фиксированных строк; 1-символьное поле с одним из двух заданных значений; и 15-байтовое поле с (равновероятно) случайными выровненными влево строками символов длиной от 4 до 15 символов.

5.1.22. *Замеры времени.* Сравните значения времени выполнения MSD-сортировки строк и трехчастной быстрой сортировки строк, используя различные генераторы ключей. Для ключей фиксированной длины добавьте LSD-сортировку.

5.1.23. *Обращения к массиву.* Сравните количество обращений к массиву, которые выполняют MSD-сортировка строк и трехчастная сортировка строк, используя различные генераторы ключей. Для ключей фиксированной длины добавьте LSD-сортировку.

5.1.24. *Самый правый просматриваемый символ.* Сравните позицию самого правого символа, который просматривают MSD-сортировка строк и трехчастная сортировка строк, используя различные генераторы ключей.

5.2. Trie-деревья

Как и в случае сортировки, свойства строк можно использовать для разработки методов поиска (реализаций таблиц имен), которые более эффективны, чем методы общего назначения из главы 3 для типичных приложений, если ключами поиска являются строки.

Методы, которые мы рассмотрим в данном разделе, обладают следующими характеристиками производительности в типичных приложениях, даже для очень больших таблиц.

- Попадания требуют времени, пропорционального длине искомого ключа.
- Для обнаружения промаха необходим просмотр лишь нескольких символов.

Такие характеристики весьма примечательны — это великолепное достижение алгоритмической теории и основной фактор, позволяющий разрабатывать вычислительную инфраструктуру, благодаря которой мы можем сейчас получить мгновенный доступ к огромным объемам информации. Более того, в API таблицы имен можно добавить операции на основе просмотра отдельных символов для строковых ключей (но не обязательно для всех ключей типа `Comparable`), которые мощны и весьма полезны на практике, как показано на рис. 5.2.1.

public class **StringST<Value>**		
	`StringST()`	*создание таблицы имен*
`void`	`put(String key, Value val)`	*занесение пары ключ-значение в таблицу (удаление `key`, если значение равно `null`)*
`Value`	`get(String key)`	*значение, соответствующее ключу `key` (`null`, если `key` отсутствует)*
`void`	`delete(String key)`	*удаление ключа **key** (и соответствующего значения)*
`boolean`	`contains(String key)`	*имеется ли значение, связанное с ключом `key`?*
`boolean`	`isEmpty()`	*пуста ли таблица?*
`String`	`longestPrefixOf(String s)`	*самый длинный ключ, который является префиксом `s`*
`Iterable<String>`	`keysWithPrefix(String s)`	*все ключи с префиксом `s` (точки означают любой символ)*
`Iterable<String>`	`keysThatMatch(String s)`	*основание (количество символов в алфавите)*
`int`	`size()`	*количество пар ключ-значение*
`Iterable<String>`	`keys()`	*все ключи из таблицы*

Рис. 5.2.1. *API для таблицы имен со строковыми ключами*

Этот API отличается от API из главы 3 следующими аспектами.

- Обобщенный тип Key заменен конкретным типом String.
- Добавлены три новых метода: longestPrefixOf(), keysWithPrefix() и keysThatMatch().

Мы будем придерживаться основных соглашений по реализациям таблиц имен из главы 3 (нет одинаковых и нулевых ключей и нет нулевых значений).

Как мы уже видели при сортировке строковых ключей, часто весьма важно иметь возможность работать со строками из заданного алфавита. Простые и эффективные реализации, удобные для небольших алфавитов, оказываются бесполезными для крупных алфавитов, т.к. они требуют слишком много памяти. В таких случаях, несомненно, удобно добавить конструктор, который позволяет клиентам указать алфавит. Мы рассмотрим реализацию такого конструктора ниже в данном разделе, но пока не включаем в наш API, чтобы не отвлекаться от строковых ключей.

В приведенных ниже описаниях трех новых методов в качестве примера используются ключи she sells sea shells by the sea shore.

- Метод longestPrefixOf() принимает в качестве аргумента строку и возвращает самый длинный ключ в таблице имен, который является префиксом этой строки. В нашем случае вызов longestPrefixOf("shell") возвращает строку she, а longestPrefixOf("shellsort") — строку shells.
- Метод keysWithPrefix() принимает в качестве аргумента строку и возвращает все ключи в таблице имен, которые являются префиксами этой строки. В нашем случае вызов keysWithPrefix("she") возвращает строки she и shells, а keysWithPrefix("se") — строки sells и sea.
- Метод keysThatMatch() принимает в качестве аргумента строку и возвращает все ключи в таблице имен, которые совпадают с этой строкой, причем точка (.) в строке аргумента соответствует любому символу. В нашем случае вызов keysThatMatch(".he") возвращает строки she и the, а keysThatMatch("s..") — строки she и sea.

Реализации и примеры применения этих операций мы рассмотрим после базовых методов таблицы имен. Они демонстрируют, что можно делать со строковыми ключами; несколько других возможностей будут рассмотрены в упражнениях.

Чтобы не отвлекаться от основных идей, мы будем заниматься методами put(), get() и новыми методами и будем считать (как в главе 3) действующими стандартные реализации contains() и isEmpty(), а реализации методов size() и delete() оставим в качестве упражнений. Поскольку строки реализуют интерфейс Comparable, можно (и нужно) включить в API упорядоченные операции, определенные в API для упорядоченной таблиц имен из главы 3. Эти реализации (обычно весьма несложные) мы также оставляем в качестве упражнений или выносим в код на сайте книги.

Trie-деревья

В данном разделе мы рассмотрим дерево поиска, которое называется *trie* — это структура данных, построенная из символов строковых ключей, которая позволяет использовать символы искомого ключа для управления поиском. Название "trie" придумал Э. Фрэдкин (E. Fredkin) в 1960 г., т.к. эта структура данных используется для выбор-

ки (re*trie*val). Мы начнем с высокоуровневого описания базовых свойств trie-деревьев, включая и алгоритмы поиска и вставки, а затем перейдем к деталям представления и реализации в Java.

Базовые свойства

Как и в случае деревьев поиска, trie-деревья представляют собой структуры данных, которые состоят из *узлов*, содержащих *ссылки* — или *нулевые*, или ссылки на другие узлы. На каждый узел указывает только один другой узел, который называется его *родителем* — кроме одного *корневого* узла, на который не указывает ни один узел — и каждый узел содержит *R* ссылок, где *R* — размер алфавита. Как правило, trie-деревья содержат значительное количество нулевых ссылок, и поэтому на графическом изображении мы будем опускать их. Хотя ссылки указывают на узлы, можно считать, что каждая ссылка указывает на trie-дерево, в корне которого находится указываемый узел. Каждая ссылка соответствует значению символа, и поскольку каждая ссылка указывает в точности на один узел, мы будем помечать узлы значениями символов, которые соответствуют указывающим на них ссылкам (кроме корня, на который не указывает ни одна ссылка). Каждый узел также содержит соответствующее *значение*, которое может быть нулевым или реальным значением, связанным с одним из строковых ключей из таблицы имен. Конкретнее, мы храним значение, связанное с каждым ключом, в узле, соответствующем его последнем символу. Очень важно помнить, что *узлы с нулевыми значениями существуют для обеспечения поиска в trie-дереве и не соответствуют никаким ключам*. Пример trie-дерева приведен на рис. 5.2.2.

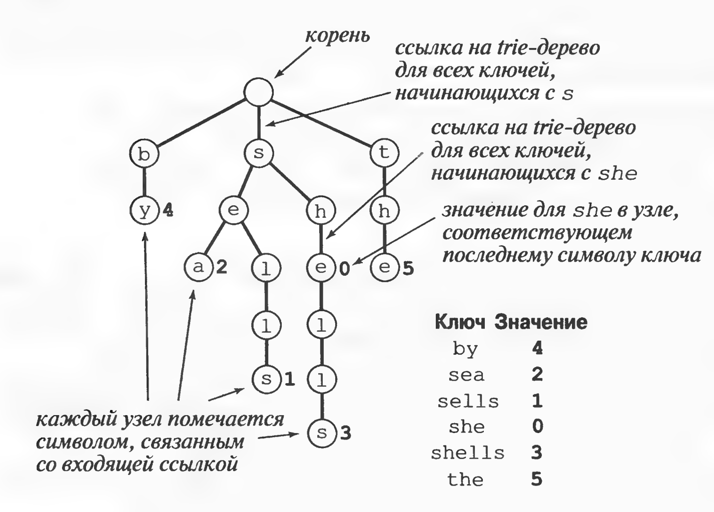


Рис. 5.2.2. Структура trie-дерева

Поиск в trie-дереве

Поиск в trie-дереве значения, связанного с заданным строковым ключом — простой процесс, который управляется символами искомого ключа. Каждый узел в trie-дереве содержит ссылку, соответствующую каждому возможному символу строки. Поиск начинается с корня, а затем следует по ссылке, связанной с первым символом ключа; от этого узла проходит по ссылке, связанной со вторым символом ключа; оттуда — по ссылке, связанной с третьим символом ключа; и т.д. до достижения последнего символа ключа

или нулевой ссылки. В этот момент возможно одно из следующих трех состояний (см. примеры на рис. 5.2.3).

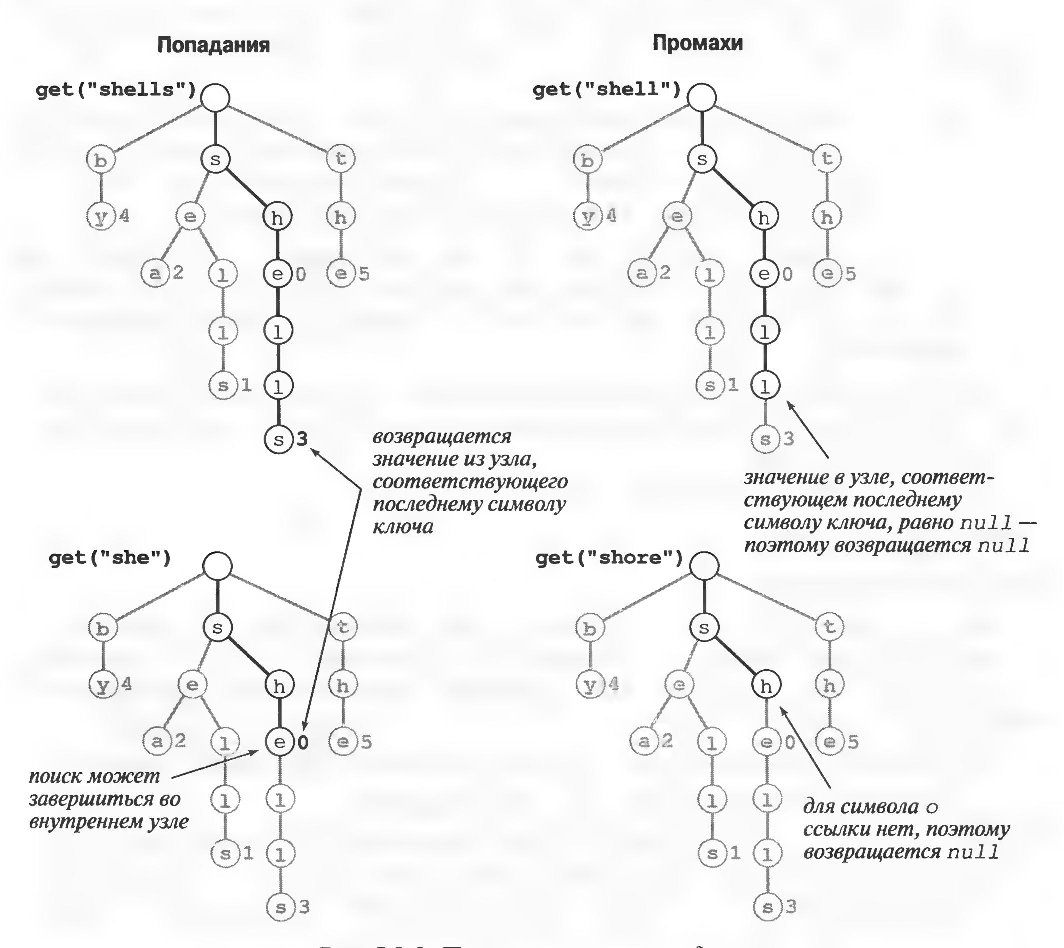


Рис. 5.2.3. Примеры поиска в trie-дереве

- Значение в узле, соответствующем последнему символу ключа, не равно `null` (как в поисках строк `shells` и `she` в левой части рис. 5.2.3). Это *попадание*: значение, связанное с ключом, равно значению в узле, соответствующем его последнему символу.
- Значение в узле, соответствующем последнему символу ключа, равно `null` (как в поиске строки `shell` вверху справа на рис. 5.2.3). Это *промах*: ключ отсутствует в таблице.
- Поиск закончился на нулевой ссылке (как в поиске строки `shore` внизу справа на рис. 5.2.3). Это тоже промах.

В любом случае для выполнения поиска нужно просто просматривать узлы по пути от корня до некоторого узла в trie-дереве.

Вставка в trie-дерево

Как и в случае бинарных деревьев поиска, для вставки нужно сначала выполнить поиск в trie-дереве, при котором символы ключа используются для спуска по дереву до достижения последнего символа ключа или нулевой ссылки. В этот момент выполнено одно из двух условий.

- Нулевая ссылка обнаружена до выборки последнего символа ключа. Это значит, что в trie-дереве нет узла, соответствующего последнему символу ключа, и необходимо создавать узлы для каждого из отсутствующих символов и в последнем созданном узле указать значение, связанное со вставляемым ключом.
- Последний символ ключа обнаружен без выхода на нулевую ссылку. В этом случае надо занести в значение узла то значение, которое связывается с ключом (независимо от того, нулевое ли оно), как обычно в нашем соглашении по ассоциативным массивам.

Во всех случаях в trie-дереве просматриваются или создаются узлы для каждого символа ключа. Создание trie-дерева для нашего стандартного клиента индексации из главы 3 с входными данными

```
she sells sea shells by the sea shore
```

показано на рис. 5.2.4.

Представление узлов

Как уже было сказано, наши графические изображения trie-деревьев не вполне соответствуют структурам данных, которые строят наши программы: на них не изображены нулевые ссылки. Учет нулевых ссылок подчеркивает следующие важные характеристики trie-деревьев.

- Каждый узел содержит *R* ссылок, по одной для каждого возможного символа.
- Символы и ключи *неявно* хранятся в структуре данных.

Например, на рис. 5.2.5 показано trie-дерево для ключей из строчных букв, где каждый узел содержит значение и 26 ссылок. Первая ссылка указывает на поддерево ключей, начинающихся с буквы `a`, второе — на поддерево ключей, начинающихся с буквы `b`, и т.д.

Ключи в trie-дереве неявно представляются путями от корня, которые заканчиваются ненулевыми значениями. Например, в trie-дереве на рис. 5.2.4 со строкой `sea` связано значение `2`, т.к. 19-я ссылка в корневом узле (которая указывает на trie-дерево для всех ключей, начинающихся с буквы `s`) не равна `null`, 5-я ссылка в указанном узле (которая указывает на trie-дерево для всех ключей, начинающихся с букв `se`) тоже не равна `null`, и первая ссылка в указанном узле (которая указывает на trie-дерево для всех ключей, начинающихся с букв `sea`) содержит значение `2`. В структуре данных нет ни строки `sea`, ни ее отдельных букв — только ссылки и значения. В силу важной роли параметра *R* мы будем называть trie-дерево для *R*-символьного алфавита ***R-частным trie-деревом***.

После этой подготовки реализация таблицы имен `TrieST` в листинге 5.2.1 уже не содержит непонятных мест. В ней используются рекурсивные методы наподобие тех, которые применялись для деревьев поиска в главе 3, и эти методы основаны на приватном классе `Node` с переменной экземпляров `val` для хранения клиентских значений и массивом `next[]` ссылок на экземпляры `Node`.

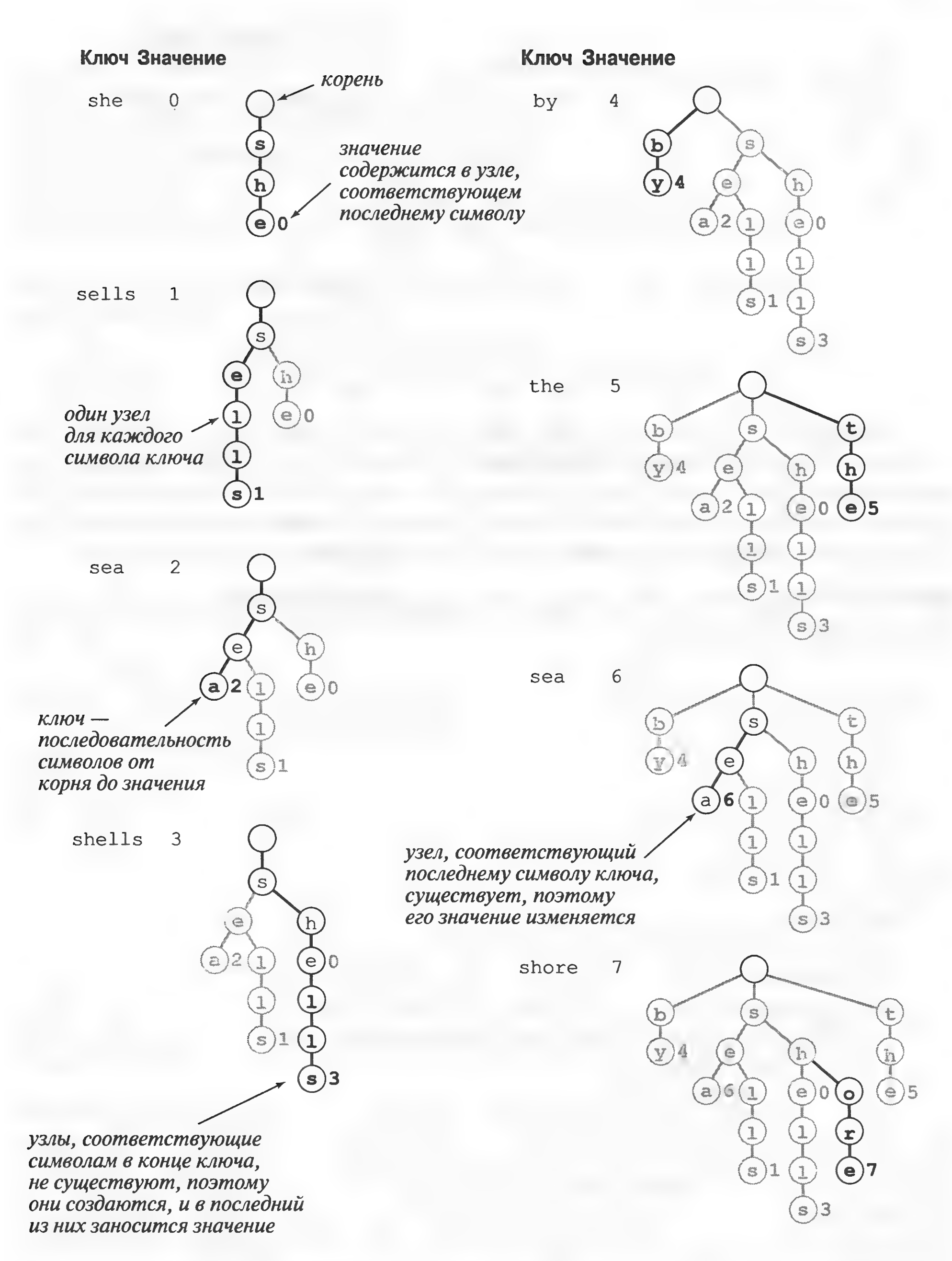


Рис. 5.2.4. Трассировка построение trie-дерева для стандартного клиента индексации

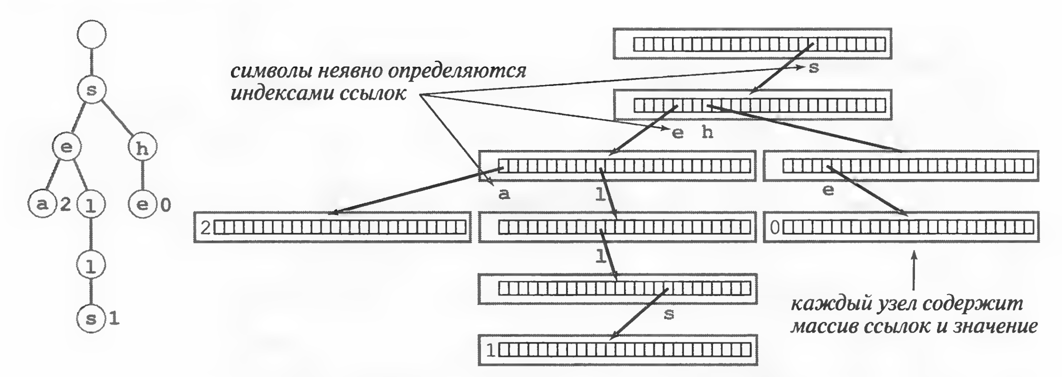


Рис. 5.2.5. Представление trie-дерева (R = 26)

Компактные рекурсивные реализации этих методов заслуживают внимательного изучения. Ниже мы рассмотрим реализации конструктора, который принимает в качестве аргумента объект `Alphabet`, и методов `size()`, `keys()`, `longestPrefixOf()`, `keysWithPrefix()`, `keysThatMatch()` и `delete()`. Это тоже вполне понятные рекурсивные методы, каждый из которых немного сложнее, чем предыдущий.

Листинг 5.2.1. Алгоритм 5.4. Таблица имен на основе trie-дерева

```
public class TrieST<Value>
{
   private static int R = 256;          // основание
   private Node root;                   // корень trie-дерева
   private static class Node
   {
     private Object val;
     private Node[] next = new Node[R];
   }
   public Value get(String key)
   {
     Node x = get(root, key, 0);
     if (x == null) return null;
     return (Value) x.val;
   }
   private Node get(Node x, String key, int d)
   { // Возвращает значение, связанное с ключом key в поддереве с корнем x.
     if (x == null) return null;
     if (d == key.length()) return x;
     char c = key.charAt(d); //Определение поддерева на основе d-го символа ключа key
     return get(x.next[c], key, d+1);
   }
   public void put(String key, Value val)
   { root = put(root, key, val, 0);  }
   private Node put(Node x, String key, Value val, int d)
   { // Изменение значения, связанного с ключом key в поддереве с корнем x.
     if (x == null) x = new Node();
     if (d == key.length()) {  x.val = val; return x; }
     char c = key.charAt(d); //Определение поддерева на основе d-го символа ключа key
     x.next[c] = put(x.next[c], key, val, d+1);
     return x;
   }
}
```

Этот код реализует таблицу имен на основе *R*-частного trie-дерева. Дополнительные методы из API таблиц строковых имен (см. рис. 5.2.1) приведены ниже. Изменение этого кода для работы с ключами из специализированных алфавитов очевидно. Значение в узле `Node` должно иметь тип `Object`, т.к. Java не поддерживает массивы обобщенных типов. Значения снова приводятся к типу `Value` в методе `get()`.

Размер

Как и в случае деревьев бинарного поиска из главы 3, для реализации метода `size()` возможны три очевидных варианта.

- "Энергичная" реализация, где количество ключей хранится в переменной экземпляров `N`.
- "Очень энергичная" реализация, где количество ключей в каждом поддереве хранится в переменной экземпляров узла и обновляется после выполнения рекурсивных вызовов `put()` и `delete()`.
- "Ленивая" реализация вроде приведенной в листинге 5.2.2. Она проходит по всем узлам trie-дерева и подсчитывает количество ненулевых значений.

Листинг 5.2.2. "Ленивый" рекурсивный метод size() для trie-деревьев

```
public int size()
{  return size(root);  }
private int size(Node x)
{
   if (x == null) return 0;
   int cnt = 0;
   if (x.val != null) cnt++;
   for (char c = 0; c < R; c++)
      cnt += size(next[c]);
   return cnt;
}
```

Как в случае деревьев бинарного поиска, "ленивая" реализация интересна, но ею лучше не пользоваться, т.к. она может стать причиной потери производительности в клиентах. "Энергичные" реализации оставлены на самостоятельную проработку в упражнениях.

Сбор ключей

Поскольку символы и ключи представлены в trie-деревьях неявно, предоставить клиентам возможность перебора ключей не так-то просто. Как и в случае деревьев бинарного поиска, строковые ключи накапливаются в объекте `Queue`, но при работе с trie-деревьями нужно создавать явные представления всех строковых ключей, а не просто выбрать их из структуры данных. Это можно сделать с помощью рекурсивного приватного метода `collect()` (см. листинг 5.2.3), который похож на `size()`, но еще и собирает строку из последовательности символов на пути от корня. При каждом посещении вершины из-за вызова `collect()` с этим узлом в качестве первого аргумента второй аргумент содержит строку, связанную с этим узлом (последовательность символов на пути от корня до узла).

Листинг 5.2.3. Сбор ключей из trie-дерева

```
public Iterable<String> keys()
{  return keysWithPrefix("");  }

public Iterable<String> keysWithPrefix(String pre)
{
   Queue<String> q = new Queue<String>();
   collect(get(root, pre, 0), pre, q);
   return q;
}

private void collect(Node x, String pre, Queue<String> q)
{
   if (x == null) return;
   if (x.val != null) q.enqueue(pre);
   for (char c = 0; c < R; c++)
      collect(x.next[c], pre + c, q);
}
```

Для посещения узла связанная с ним строка заносится в очередь (если ее значение не нулевое), а затем (рекурсивно) посещаются все узлы из массива его ссылок, для каждого возможного символа. Для создания ключа при каждом вызове к текущему ключу добавляется символ, соответствующий ссылке. Этот метод `collect()` выполняет сбор ключей для методов `keys()` и `keysWithPrefix()` из нашего API. В реализации `keys()` вызывается метод `keysWithPrefix()` с пустой строкой в качестве аргумента; в реализации `keysWithPrefix()` вызывается метод `get()`, чтобы найти узел trie-дерева, соответствующий указанному префиксу (`null`, если такого узла нет), а затем для завершения работы используется метод `collect()`. На рис. 5.2.6 приведена трассировка работы метода `collect()` (или `keysWithPrefix("")`) для примера trie-дерева, где указаны значения второго аргумента и содержимое очереди при каждом вызове `collect()`.

На рис. 5.2.7 показан процесс для вызова `keysWithPrefix("sh")`.

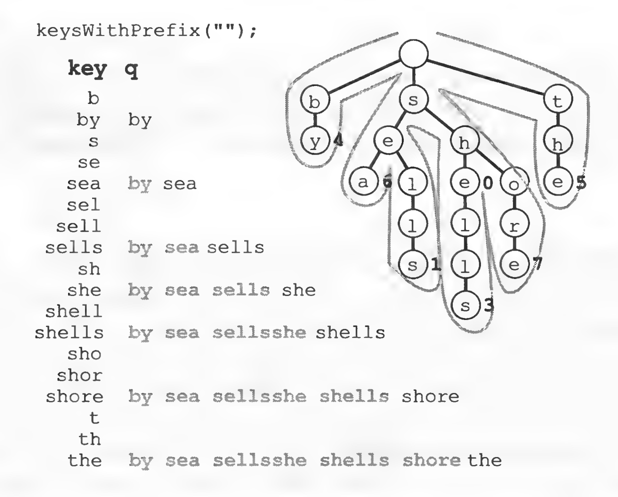


Рис. 5.2.6. Сбор ключей из trie-дерева (трассировка)

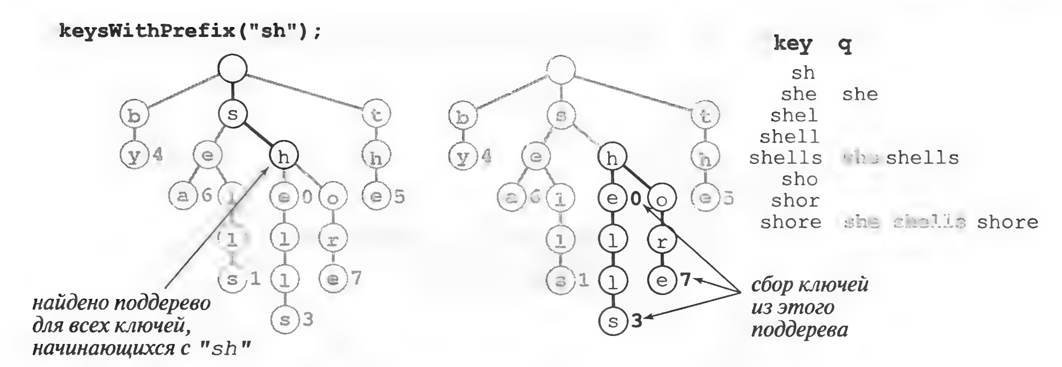


Рис. 5.2.7. Сравнение префиксов в trie-дереве

Сравнение с обобщенными символами

В реализации keysThatMatch() используется аналогичный процесс, но с добавлением аргумента, который задает шаблон для collect() и добавляет в рекурсивный вызов проверку всех ссылок, если это обобщенный символ шаблона, или только ссылки, соответствующей конкретному символу, как в коде листинга 5.2.4. Обратите также внимание, что нет необходимости рассматривать ключи, которые длиннее шаблона.

Листинг 5.2.4. Сравнение обобщенных символов в trie-дереве

```
public Iterable<String> keysThatMatch(String pat)
{
   Queue<String> q = new Queue<String>();
   collect(root, "", pat, q);
   return q;
}

public void collect(Node x, String pre, String pat, Queue<String> q)
{
   int d = pre.length();
   if (x == null) return;
   if (d == pat.length() && x.val != null) q.enqueue(pre);
   if (d == pat.length()) return;

   char next = pat.charAt(d);
   for (char c = 0; c < R; c++)
      if (next == '.' || next == c)
         collect(x.next[c], pre + c, pat, q);
}
```

Самый длинный префикс

Чтобы найти самый длинный ключ, который является префиксом заданной строки, используется рекурсивный метод наподобие get(), который отслеживает длину самого длинного ключа, найденного на пути поиска (для этого он передается в качестве параметра в рекурсивный метод и изменяется, если посещается узел с ненулевым значением). Поиск заканчивается при обнаружении конца строки или нулевой ссылки — смотря что встретится раньше (см. рис. 5.2.8 и листинг 5.2.5).

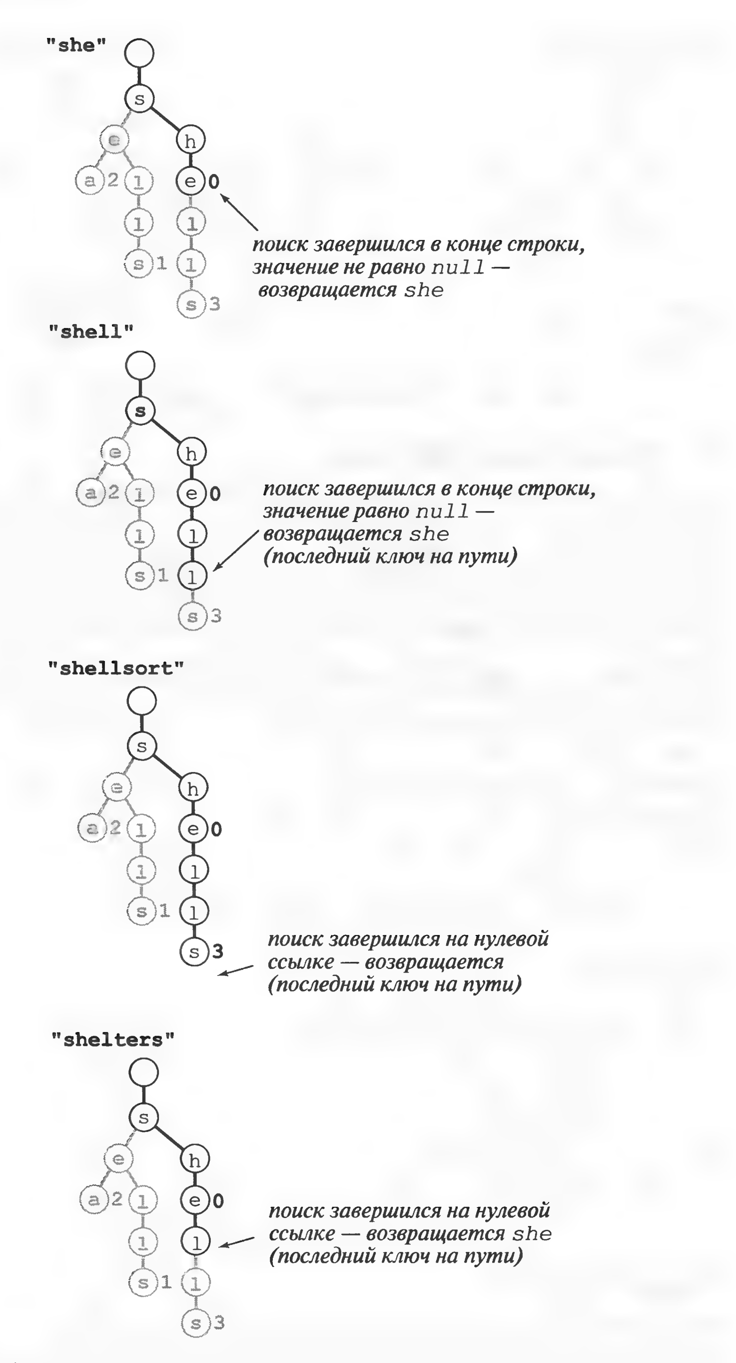


Рис. 5.2.8. Возможные ситуации при выполнении `longestPrefixOf()`

Листинг 5.2.5. Сравнение самого длинного префикса в заданной строке

```
public String longestPrefixOf(String s)
{
   int length = search(root, s, 0, 0);
   return s.substring(0, length);
}

private int search(Node x, String s, int d, int length)
{
   if (x == null) return length;
   if (x.val != null) length = d;
   if (d == s.length()) return length;
   char c = s.charAt(d);
   return search(x.next[c], s, d+1, length);
}
```

Удаление

Первым шагом при удалении из trie-дерева пары ключ-значение нужно выполнить обычный поиск, чтобы найти узел, соответствующий удаляемому ключу, и занести в соответствующее значение `null`. Если этот узел содержит хоть одну ненулевую ссылку на дочерний узел, то больше ничего делать не надо, а если все ссылки нулевые, то нужно удалить и сам этот узел из структуры данных. Если после этого в родительском узле остались только нулевые ссылки, то необходимо удалить и тот узел, и т.д (рис. 5.2.9). Реализация в листинге 5.2.6 демонстрирует, что это действие можно выполнить с помощью весьма небольшого объема кода, используя стандартную рекурсивную схему: после рекурсивных вызовов для узла `x` возвращается `null`, если клиентское значение и все ссылки в узле равны `null`, иначе возвращается `x`.

Листинг 5.2.6. Удаление ключа (и связанного с ним значения) из trie-дерева

```
public void delete(String key)
{  root = delete(root, key, 0);  }

private Node delete(Node x, String key, int d)
{
   if (x == null) return null;
   if (d == key.length())
      x.val = null;
   else
   {
      char c = key.charAt(d);
      x.next[c] = delete(x.next[c], key, d+1);
   }
```

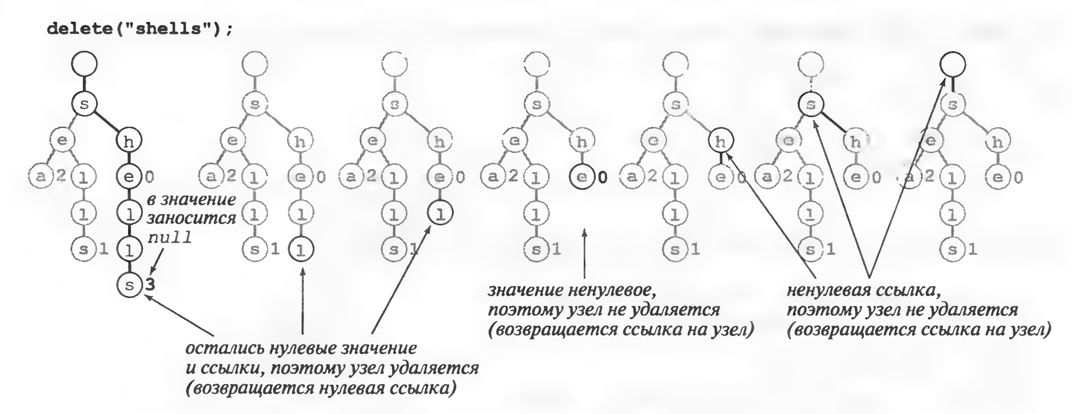


Рис. 5.2.9. Удаление ключа (и связанного с ним значения) из trie-дерева

Алфавит

Как обычно, код алгоритма 5.4 написан для ключей Java-типа `String`, но его нетрудно изменить для обработки ключей, взятых из любого алфавита. Для этого нужно выполнить перечисленные ниже действия.

- Реализовать конструктор, который принимает в качестве аргумента заданный алфавит, копирует аргумент в переменную экземпляров типа `Alphabet`, а количество символов в алфавите — в переменную `R`.
- Использовать в методах `get()` и `put()` вызов метода `Index()` из класса `Alphabet` для преобразования символов строк в индексы от 0 до $R-1$.
- Использовать метод `toChar()` из класса `Alphabet` для преобразования индексов от 0 до $R-1$ в значения `char`. Эта операция не нужна в методах `get()` и `put()`, но нужна в реализациях методов `keys()`, `keysWithPrefix()` и `keysThatMatch()`.

Если известно, что ключи берутся из небольшого алфавита, такие изменения помогают сохранить значительный объем памяти (используя лишь R ссылок в каждом узле), но за счет времени, необходимого для преобразования символов в индексы и наоборот.

Рассмотренный нами код представляет собой компактную и полную реализацию API таблицы строковых имен, который имеет много полезных практических применений. Несколько модификаций и расширений рассматриваются в упражнениях. А теперь мы рассмотрим основные свойства trie-деревьев и некоторые ограничения на их применимость.

Свойства trie-деревьев

Как обычно, нас интересует время и память, которые необходимы для применения trie-деревьев в типичных приложениях. Trie-деревья интенсивно изучались и анализировались, и их базовые свойства относительно несложно понимать и применять.

Утверждение Е. Связная структура (форма) trie-дерева не зависит от порядка вставки/удаления ключей: для любого заданного множества ключей существует уникальное trie-дерево.

Доказательство. Непосредственно по индукции по поддеревьям.

Этот фундаментальный факт является отличительной характеристикой trie-деревьев: во всех остальных древовидных структурах поиска, которые уже изучены нами, построенное дерево зависит как от множества ключей, так и от порядка их вставки.

Граница времени для поиска и вставки в худшем случае

За какое время можно найти значение, связанное с ключом? Для ответа на этот вопрос в случае деревьев бинарного поиска, хеширования и других методов из главы 4 нужен был математический анализ, но для trie-деревьев ответить на него очень легко.

Утверждение Ж. Количество обращений к массиву при поиске в trie-дереве или при вставке в него ключа не больше длины ключа плюс 1.

Доказательство. Непосредственно следует из кода. Рекурсивные реализации `get()` и `put()` содержат аргумент `d`, который вначале равен 0, потом увеличивается при каждом вызове и используется для остановки рекурсии при достижении длины ключа.

С теоретической точки зрения из утверждения Ж следует, что trie-деревья *оптимальны* в случаях попаданий: невозможно ожидать времени лучшего, чем пропорциональное длине искомого ключа. Какие бы алгоритм или структура данных ни применялись, мы не можем знать, что ключ найден, не просмотрев все его символы. С практической точки зрения эта гарантия важна тем, что *она не зависит от количества ключей*: при работе с 7-символьными номерными знаками мы знаем, что для попадания или промаха достаточно проверить не более 8 узлов; при работе с 20-значными учетными номерами для этого необходимо просмотреть не более 21 узла.

Граница ожидаемого времени для промахов

Допустим, что при поиске ключа в trie-дереве мы обнаружили, что ссылка в корневом узле, соответствующем первому символу, содержит `null`. Тогда мы сразу знаем, что ключ отсутствует в таблице, проверив лишь *один* узел. Этот случай типичен: одно их наиболее важных свойств trie-деревьев состоит в том, что для обнаружения промахов обычно достаточно просмотреть лишь несколько узлов. Если предположить, что ключи выбираются из модели случайных строк (каждый символ может появиться с той же вероятностью, что и любой другой из R различных символов), то можно доказать следующее утверждение.

Утверждение З. Среднее количество узлов, проверяемых при промахе в trie-дереве, построенного из N случайных ключей из алфавита размером R, равно $\sim\log_R N$.

Набросок доказательства. (Для читателей, знакомых с вероятностным анализом.) Вероятность, что каждый из N ключей в случайном trie-дереве отличается от случайного искомого ключа хотя бы одним из t старших символов, равна $(1-R^{-t})^N$. Вычтя эту величину из 1, получим вероятность, что один из ключей trie-дерева совпадает с искомым ключом во всех старших t символах. То есть $1-(1-R^{-t})^N$ — вероятность того, что для поиска потребуется более t сравнений символов. Из вероятностного анализа известно, что сумма вероятностей того, что целая случайная переменная больше t для $t = 0, 1, 2, \ldots$, равна среднему значению этой случайной переменной, откуда средняя стоимость поиска равна

$$1-(1-R^{-1})^N + 1-(1-R^{-2})^N + \ldots + 1-(1-R^{-t})^N + \ldots$$

Применив элементарную аппроксимацию $(1-1/x)^x \sim e^{-1}$, находим, что стоимость поиска примерно равна

$$(1-e^{-N/R^1}) + (1-e^{-N/R^2}) + \ldots + (1-e^{-N/R^t}) + \ldots$$

Здесь примерно $\log_R N$ слагаемых очень близки к 1 (в которых R^t значительно меньше N), а еще есть слагаемые, очень близкие к 0 (в которых R^t значительно больше N), и несколько слагаемых со значениями между 0 и 1 (в которых $R^t \approx N$). Поэтому общая сумма имеет порядок $\log_R N$.

С практической точки зрения наиболее важным следствием из этого утверждения является то, что *стоимость промахов не зависит от длины ключа*. Например, получается, что для неудачного поиска в trie-дереве, построенного из 1 миллиона случайных ключей, необходимо просмотреть лишь три-четыре узла — независимо от того, являются ли ключами номерные знаки из 7 символов или учетные записи из 20 цифр. Конечно, не стоит ожидать в реальности действительно случайные ключи, но *разумно* предположить, что эта модель приемлемо описывает поведение trie-алгоритмов для ключей в типичных приложениях. И действительно, такое поведение широко встречается на практике и поэтому является серьезной причиной для широкого применения trie-деревьев.

Память

Сколько памяти необходимо для trie-дерева? Ответ на этот вопрос (и оценка объема *доступной* памяти) крайне важен для эффективного использования trie-деревьев.

Утверждение И. Количество ссылок в trie-дереве находится между RN и RNw, где w — средняя длина ключей.

Доказательство. Для каждого ключа в trie-дереве имеется узел, содержащий связанное с ключом значение. Этот узел содержит также R ссылок, поэтому количество ссылок не меньше RN. Если первые символы всех ключей различаются, то для каждого символа ключа существует узел с R ссылками, поэтому количество ссылок в R раз больше общего количества символов в ключах — т.е. RNw.

В табл. 5.2.1 приведены стоимости для некоторых типичных рассмотренных нами приложений.

Таблица 5.2.1. Объемы памяти для типичных trie-деревьев

Область	Типичный ключ	Средняя длина w	Размер алфавита R	Количество ссылок в trie-дереве, построенном из 1 миллиона ключей
Автомобильные номера в Калифорнии	`4PGC938`	7	256	256 миллионов
Номера счетов	`02400019992993299111`	20	256	4 миллиарда
			10	256 миллионов
URL-адреса	`www.cs.princeton.edu`	28	256	4 миллиарда
Обработка текста	`seashells`	11	256	256 миллионов
Протеины в геномике	`ACTGACTG`	8	256	256 миллионов
			4	4 миллиона

Она демонстрирует следующие приблизительные правила для trie-деревьев.

- Если ключи короткие, количество ссылок близко к *RN*.
- Если ключи длинные, количество ссылок близко к *RNw*.
- Поэтому уменьшение *R* может сэкономить значительный объем памяти.

Не столь очевидный вывод из этой таблицы: прежде чем приступить к развертыванию trie-деревьев в приложении, важно уяснить свойства включаемых в них ключей.

Однонаправленные ветви

Основной причиной огромного объема trie-деревьев для длинных ключей является то, что длинные ключи часто приводят к появлению в trie-дереве длинных "хвостов", где каждый узел содержит единственную ссылку на следующий узел (и, следовательно, $R-1$ нулевую ссылку), как показано на рис. 5.2.10. Эту ситуацию нетрудно исправить (см. упражнение 5.2.11). В trie-дереве могут находиться и внутренние однонаправленные ветви. Например, два длинных ключа могут отличаться только одним последним символом. Такую ситуацию исправить несколько труднее (см. упражнение 5.2.12). Эти изменения могут сделать требования к памяти trie-деревьев не таким важным фактором, как для приведенной примитивной реализации, но они не обязательно эффективны в практических приложениях. Ниже мы рассмотрим альтернативный подход для уменьшения требований к памяти в trie-деревьях.

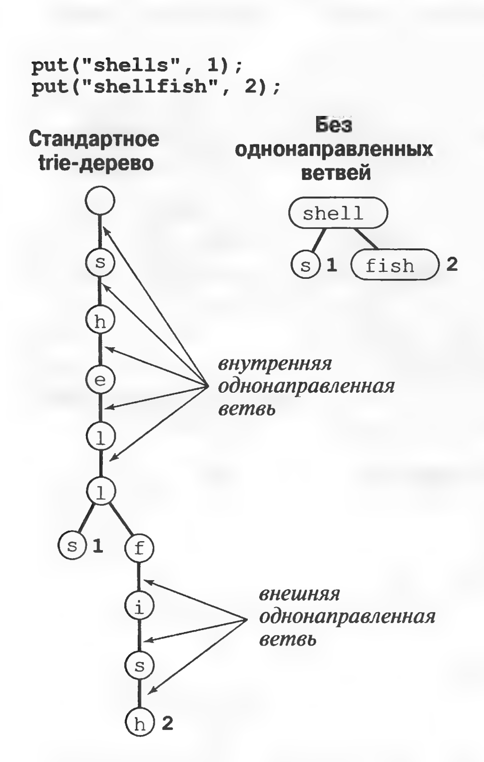


Рис. 5.2.10. Устранение однонаправленных ветвей из trie-дерева

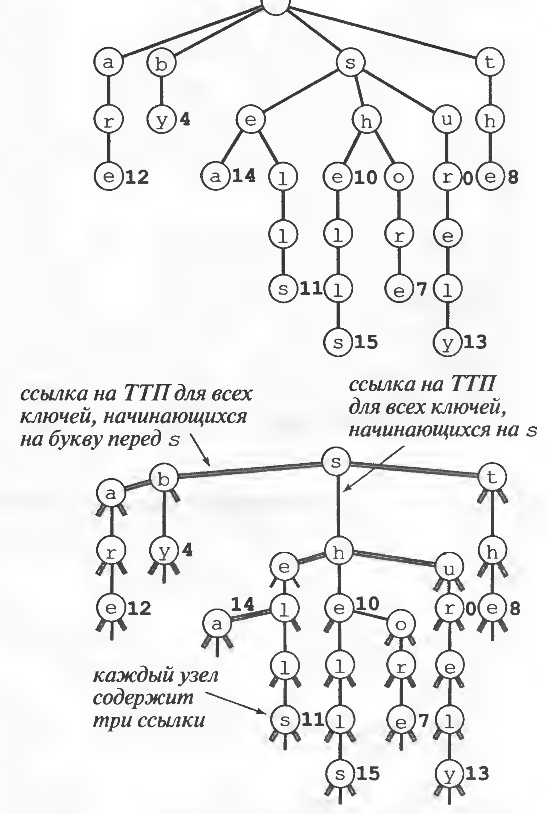


Рис. 5.2.11. Представление trie-дерева в виде ТТП

В общем, вывод таков: *не пытайтесь использовать алгоритм 5.4 для больших количеств длинных ключей из больших алфавитов*, т.к. для этого потребуется объем памяти, пропорциональный *R*, умноженному на общее количество символов в ключах. Но если необходимая память доступна, производительность trie-деревьев трудно превзойти.

Trie-деревья тернарного поиска (ТТП)

Чтобы справиться с огромными запросами памяти у *R*-частных trie-деревьев, сейчас мы рассмотрим альтернативное представление — trie-дерево тернарного поиска (ternary search trie). В ТТП каждый узел содержит символ, *три* ссылки и значение (рис. 5.2.11). Эти три ссылки соответствуют ключам, текущие символы которых меньше, равны или больше символа данного узла. В *R*-частных trie-деревьях из алгоритма 5.4 узлы этих деревьев представлены *R* ссылками, причем каждый символ соответствует каждой ненулевой ссылке и неявно представлен его индексом. В соответствующем ТТП символы *явно* присутствуют в узлах, и эти символы можно обнаружить при переходах по средним ссылкам.



Рис. 5.2.12. Пример поиска в ТТП

Поиск и вставка

Код поиска и вставки для реализации API таблицы имен на основе ТТП очевиден (рис. 5.2.12). Для поиска выполняются сравнения первого символа ключа с символом в корне. Если он меньше, выбирается левая ссылка, если больше — правая, а если равен, выбирается средняя ссылка, и мы переходим к следующему символу искомого ключа. В любом случае алгоритм применяется рекурсивно. Он завершает работу *промахом*, если обнаружена нулевая ссылка или если узел, на котором завершился поиск, содержит нулевое значение; он завершает работу *попаданием*, если узел, на котором завершился поиск, содержит ненулевое значение. Чтобы вставить новый ключ, выполняется его поиск, а затем добавляются новые узлы для символов в конце ключа — как в trie-деревьях. Детали реализации этих методов приведены в алгоритме 5.5 (листинг 5.2.7).

Листинг 5.2.7. Алгоритм 5.5. Таблица имен на основе ТТП

```
public class TST<Value>
{
   private Node root;                  // корень дерева

   private class Node
   {
      char c;                          // символ
      Node left, mid, right;           // левое, среднее и правое поддеревья
      Value val;                       // значение, связанное со строкой
   }
   public Value get(String key)  // как и для trie-деревьев (см. листинг 5.2.1)
```

```
  private Node get(Node x, String key, int d)
  {
     if (x == null) return null;
     char c = key.charAt(d);
     if      (c < x.c) return get(x.left,  key, d);
     else if (c > x.c) return get(x.right, key, d);
     else if (d < key.length() - 1)
                  return get(x.mid,   key, d+1);
     else return x;
  }

  public void put(String key, Value val)
  {  root = put(root, key, val, 0);  }

  private Node put(Node x, String key, Value val, int d)
  {
     char c = key.charAt(d);
     if (x == null)  { x = new Node(); x.c = c; }
     if      (c < x.c) x.left  = put(x.left,  key, val, d);
     else if (c > x.c) x.right = put(x.right, key, val, d);
     else if (d < key.length() - 1)
                    x.mid   = put(x.mid,   key, val, d+1);
     else x.val = val;
     return x;
  }
}
```

В этой реализации каждый узел содержит значение `c` типа `char` и три ссылки — это дает возможность строить trie-деревья поиска строк, где поддеревья соответствуют ключам, первый символ которых меньше `c` (левое), равен `c` (среднее) и больше `c` (правое) (рис. 5.2.13).

Эта схема эквивалентна реализации каждого узла *R*-частного trie-дерева в виде дерева бинарного поиска, в котором в качестве ключей используются символы, соответствующие ненулевым ссылкам. В отличие от этого, в алгоритме 5.4 задействован индексированный ключами массив. ТТП и соответствующее ему trie-дерево приведены на рис. 5.2.11. Продолжая сравнение деревьев бинарного поиска и алгоритмов сортировки (глава 3), мы видим, что ТТП соответствуют быстрой сортировке, а trie-деревья соответствуют MSD-сортировке.

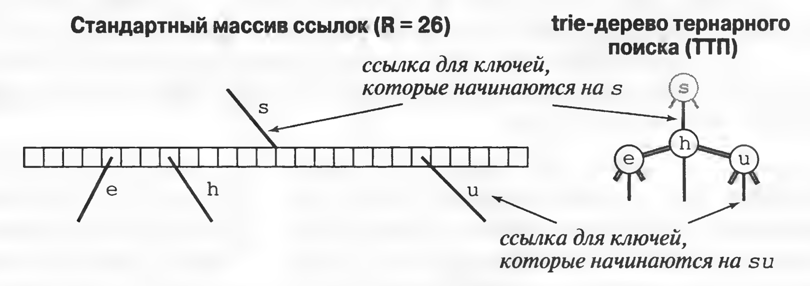


Рис. 5.2.13. Представления узлов trie-дерева

Рисунки 5.1.12 и 5.1.17, где показаны структуры рекурсивных вызовов для MSD-сортировки и трехчастной быстрой сортировки строк (соответственно), в точности соответствуют trie-дереву и ТТП с рис. 5.2.11 для этого множества ключей. Память для ссылок в trie-деревьях соответствует памяти для счетчиков в сортировке строк: трехчастное ветвление обеспечивает эффективное решение обеих задач.

Свойства ТТП

ТТП является компактным представлением *R*-частного trie-дерева, однако свойства этих двух структур данных существенно различаются. Пожалуй, наиболее важное отличие состоит в том, что для ТТП свойство А не выполняется: представления любого узла trie-дерева с помощью дерева бинарного поиска зависит от порядка вставки ключей, как и в любом другом дереве бинарного поиска.

Память

Самое важное свойство ТТП — то, что они содержат в каждом узле лишь по три ссылки, поэтому ТТП занимают меньше памяти, чем соответствующее trie-дерево.

Утверждение К. ТТП, построенное из N строковых ключей средней длины w содержит от $3N$ до $3Nw$ ссылок.

Доказательство. Аналогично рассуждению для утверждения И.

Реальный объем занимаемой памяти обычно меньше верхней границы в три ссылки на символ, т.к. ключи с общими префиксами совместно используют узлы на верхних уровнях дерева.

Стоимость поиска

Чтобы определить стоимость поиска (и вставки) в ТТП, нужно умножить стоимость для соответствующего trie-дерева на стоимость прохода по ТТП-представлению каждого узла trie-дерева.

Утверждение Л. Для обнаружения промаха в ТТП, построенном из N случайных строковых ключей, требуется в среднем $\sim \ln N$ сравнений символов. Для попадания или вставки в ТТП выполняется сравнение для каждого символа искомого ключа.

Доказательство. Стоимость попадания/вставки непосредственно следует из кода. Стоимость промаха вытекает из рассуждений, аналогичных приведенным в наброске доказательства утверждения З. Мы предполагаем, что все узлы на пути поиска, кроме некоторого константного их количества (несколько узлов на верхних уровнях) ведут себя как случайные деревья бинарного поиска с R значениями символов со средней длиной пути $\ln R$, поэтому временную стоимость $\log_R N = \ln N / \ln R$ нужно умножить на $\ln R$.

В худшем случае узел может быть представлен несбалансированным полным R-частным деревом, вытянутым в виде единого связного списка, и тогда стоимость нужно умножить на R. Однако обычно можно ожидать $\ln R$ или менее сравнений символов (поскольку корневой узел ведет себя как случайное дерево бинарного поиска для R различных значений символов) на верхнем уровне и, возможно, на нескольких других уровнях — при наличии ключей с общим префиксом и до R различных значений

для символа, следующего за префиксом. Кроме того, обычно для большинства символов также выполняется лишь несколько сравнений, т.к. большинство trie-узлов содержат немного ненулевых ссылок. При промахах, скорее всего, будет выполнено лишь несколько сравнений символов с завершающей нулевой ссылкой где-то на верхних уровнях trie-дерева, а при попаданиях выполняется примерно только по одному сравнению на каждый символ искомого ключа, потому что большинство из них находятся в узлах с одной исходящей ветвью на нижних уровнях trie-дерева.

Алфавит

Основное преимущество использования ТТП состоит в том, что они гибко приспосабливаются к нерегулярностям в искомых ключах, которые часто встречаются в практических приложениях. Обратите, в частности, внимание на то, что в этом случае уже нет необходимости разрешать построение строк из клиентского алфавита, что было важно для trie-деревьев. В данном случае важны два главных эффекта. Во-первых, ключи в практических приложениях берутся из больших алфавитов, и частота отдельных символов далека от равномерного распределения. В ТТП можно использовать 256-символьную кодировку ASCII или 65536-символьную кодировку Unicode, не заботясь о громадной стоимости узлов с 256- или 65536-путевым ветвлением, и даже не заботясь об определении конкретного набора символов. Строки в кодировке Unicode для некоторых восточных алфавитов могут содержать тысячи различных символов — ТТП особенно удобны для стандартных ключей типа `String`, состоящих из таких символов.

Во-вторых, в практических приложениях ключи часто имеют структурированный вид (различный в разных приложениях) — например, в одной части ключа допустимы только буквы, а в другой только цифры. В нашем примере с калифорнийскими автомобильными номерами второй, третий и четвертый символы должны быть прописными латинскими буквами ($R = 26$), а остальные символы — цифрами ($R = 10$). В ТТП для таких ключей некоторые trie-узлы представляют собой дерево бинарного поиска с 10 узлами (для позиций, в которых допустимы только цифры), а другие представляют собой дерево бинарного поиска с 26 узлами (для позиций, в которых допустимы только заглавные буквы). Эта структура формируется автоматически, без необходимости специального анализа ключей.

Совпадение префиксов, сбор ключей и совпадение с обобщенными символами

Поскольку ТТП представляют trie-деревья, реализации методов `longestPrefixOf()`, `keys()`, `keysWithPrefix()` и `keysThatMatch()` легко получить адаптацией соответствующего кода для trie-деревьев из предыдущего раздела — это полезное упражнение для закрепления понимания и trie-деревьев, и деревьев тернарного поиска (см. упражнение 5.2.9). При этом имеет место тот же компромисс, что и для поиска: линейный объем необходимой памяти, но дополнительный множитель $\ln R$ на каждое сравнение символов.

Удаление

При реализации метода `delete()` для ТТП придется потрудиться больше. Ведь каждый символ удаляемого ключа принадлежит некоторому дереву бинарного поиска. В trie-дереве можно удалить ссылку, соответствующую символу, занеся в соответствующий элемент массива ссылок значение `null`; но в ТТП для удаления узла, соответствующего какому-то символу, придется использовать удаление узла из дерева бинарного поиска.

Гибридные ТТП

Возможно легкое усовершенствование поиска на основе ТТП — использование большого явного многочастного узла в корне. Для этого проще всего задействовать таблицу из R деревьев тернарного поиска, по одному для каждого возможного значения первого символа ключей. Если R не очень велико, можно использовать и первые два символа ключей (и таблицу размером R^2). Для эффективности этого метода нужно, чтобы старшие символы ключей были довольно равномерно распределены. Полученный гибридный алгоритм поиска похож на то, как люди ищут имена в телефонной книге. Сначала выполняется многочастное ветвление (“Так... начинается на `А`”), потом принимаются бинарные решения (“Перед `Андреев`, но после `Айтматов`”), а затем выполняется последовательное сравнение (“`Алкоголь`, ... Ага, слова `Алконавт` здесь нет, потому что нет слов, начинающихся на `Алкон`”). Эти программы находятся в лидерах быстрого поиска строковых ключей.

Однонаправленные ветви

Как и в случае trie-деревьев, можно повысить эффективность ТТП, если поместить ключи в листья в те моменты, когда их можно отличить от других, и таким образом устранить однонаправленные ветви из многих внутренних узлов.

Утверждение М. Поиск или вставка в ТТП, построенном из N случайных строковых ключей, без внешнего одностороннего ветвления и с R^t-путевым ветвлением в корне требует в среднем примерно $\ln N - t\ln R$ сравнений.

Доказательство. Эти грубые оценки следуют из рассуждения, аналогичного тому, которое было использовано для доказательства утверждения Л. Мы предполагаем, что все узлы на пути поиска, кроме константного количества (несколько в самом верху дерева) выполняют функции деревьев бинарного поиска с R значениями символов, поэтому временные затраты умножаются на $\ln R$.

Несмотря на естественное желание настроить алгоритм на максимальную производительность, не следует забывать, что одной из наиболее привлекательных черт ТТП является их независимость от конкретных характеристик приложения, в силу чего они обеспечивают хорошую производительность и без всякой настройки.

Какую реализацию таблицы символьных имен следует использовать?

Как и в случае сортировки строк, естественно поинтересоваться, как рассмотренные здесь методы поиска строк соотносятся с методами общего назначения, рассмотренными в главе 3. В табл. 5.2.2 приведены важные характеристики алгоритмов, которые были рассмотрены в данном разделе (строки для деревьев бинарного поиска, красно-черных деревьев бинарного поиска и хеширования взяты для сравнения из главы 3). Для любого конкретного приложения эту таблицу следует рассматривать как ориентировочную, т.к. при изучении реализаций таблиц имен необходимо учитывать очень много факторов (таких как характеристики ключей и смеси операций с ними).

Таблица 5.2.2. Характеристики производительности алгоритмов поиска строк

Алгоритм (структура данных)	Типичная скорость роста для N строк из R-символьного алфавита (средняя длина w)		Сильные стороны
	Количество символов, просматриваемых при промахе	Объем необходимой памяти	
Дерево бинарного поиска	$c_1\ (\lg N)^2$	$64N$	Случайно упорядоченные ключи
Поиск в 2-3-дереве (красно-черное дерево бинарного поиска)	$c_2\ (\lg N)^2$	$64N$	Гарантированная производительность
Линейное опробование (параллельные массивы)	w	от $32N$ до $128N$	Встроенные типы, кешированные хеш-значения
Поиск в trie-дереве (R-частное trie-дерево)	$\log_R N$	от $(8R+56)N$ до $(8R+56)Nw$	Короткие ключи, небольшие алфавиты
Поиск в trie-дереве (ТТП)	$1{,}39\ \lg N$	от $64N$ до $64Nw$	Неслучайные ключи

При наличии необходимого объема памяти R-частные trie-деревья обеспечивают самый быстрый поиск, выполняя для этого *константное* количество сравнений символов. Для больших алфавитов, когда памяти для R-частных деревьев может не хватить, удобнее ТТП, поскольку они используют логарифмическое количество сравнений *символов*, в то время как деревья бинарного поиска используют логарифмическое количество сравнений *ключей*. Иногда весьма удобно хеширование, но, как обычно, оно не может поддерживать операции упорядоченной таблицы имен или дополнительные операции, основанные на работе с отдельными символами, такие как сравнение префиксов или обобщенных символов.

Вопросы и ответы

Вопрос. **Используется ли один из этих методов в системной сортировке Java для поиска ключей типа `String`?**

Ответ. Нет.

Упражнения

5.2.1. Нарисуйте R-частное trie-дерево, полученное после вставки ключей

`no is th ti fo al go pe to co to th ai of th pa`

в указанном порядке в первоначально пустое trie-дерево (нулевые ссылки не изображайте).

5.2.2. Нарисуйте ТТП, полученное после вставки ключей

`no is th ti fo al go pe to co to th ai of th pa`

в указанном порядке в первоначально пустое ТТП.

5.2.3. Нарисуйте R-частное trie-дерево, полученное после вставки ключей

`now is the time for all good people to come to the aid of`

в указанном порядке в первоначально пустое trie-дерево (нулевые ссылки не изображайте).

5.2.4. Нарисуйте ТТП, полученное после вставки ключей

`now is the time for all good people to come to the aid of`

в указанном порядке в первоначально пустое ТТП.

5.2.5. Напишите нерекурсивные версии реализаций `TrieST` и `TST`.

5.2.6. Реализуйте API для типа данных `StringSET`, приведенный на рис. 5.2.14.

public class **StringSET**		
	`StringSET()`	*создание множества строк*
`void`	`add(String key)`	*занесение в множество ключа* `key`
`void`	`delete(String key)`	*удаление ключа* `key` *из множества*
`boolean`	`contains(String key)`	*присутствует ли ключ* `key` *в множестве?*
`boolean`	`isEmpty()`	*пусто ли множество?*
`int`	`size()`	*количество ключей в множестве*
`String`	`toString()`	*строковое представление множества*

Рис. 5.2.14. *API для типа данных множества строк*

Творческие задачи

5.2.7. *Пустые строки в ТТП.* Код реализации ТТП неправильно обрабатывает пустые строки. Объясните, в чем дело, и предложите способ устранения этой проблемы.

5.2.8. *Упорядоченные операции для trie-деревьев.* Реализуйте методы `floor()`, `ceil()`, `rank()` и `select()` (из нашего стандартного API для упорядоченной таблицы поиска из главы 3) для класса `TrieST`.

5.2.9. *Расширенные операции для ТТП.* Реализуйте метод `keys()` и расширенные операции, описанные в данном разделе — `longestPrefixOf()`, `keysWithPrefix()` и `keysThatMatch()` — для класса `TST`.

5.2.10. *Размер.* Реализуйте “очень энергичный” метод `size()` (который хранит в каждом узле количество ключей в поддереве этого узла) для классов `TrieST` и `TST`.

5.2.11. *Внешние однонаправленные ветви.* Добавьте в реализации `TrieST` и `TST` код для устранения внешних односторонних ветвей.

5.2.12. *Внутренние однонаправленные ветви.* Добавьте в реализации `TrieST` и `TST` код для устранения внутренних односторонних ветвей.

5.2.13. *Гибридное ТТП с R^2-частным ветвлением в корне.* Добавьте в реализацию `TST` код для выполнения многочастного ветвления на первых двух уровнях, как описано в тексте.

5.2.14. *Уникальные подстроки длиной L.* Напишите клиент `TST`, который вводит текст из стандартного ввода и подсчитывает в нем количество уникальных подстрок длиной *L*. Например, входной текст `cgcgggcgcg` содержит пять уникальных подстрок длиной 3 — `cgc`, `cgg`, `gcg`, `ggc` и `ggg`. *Совет*: выберите `i`-ю подстроку с помощью вызова `substring(i, i + L)` и вставьте ее в таблицу имен.

5.2.15. *Уникальные подстроки.* Напишите клиент `TST`, который вводит текст из стандартного ввода и подсчитывает в нем количество уникальных подстрок любых длин. Это можно очень эффективно выполнить с помощью дерева суффиксов — см. главу 6.

5.2.16. *Сходство документов.* Напишите клиент `TST` со статическим методом, который принимает из командной строки целочисленное значение `L` и два имени файлов и вычисляет `L`-сходство двух документов — евклидово расстояние между векторами частот, которые определяются как количество каждой триграммы в документе, деленное на количество триграмм. Напишите статический метод `main()`, который принимает в качестве аргумента командной строки целое значение `L`, а из стандартного ввода список имен файлов, и выводит матрицу `L`-сходства всех пар документов.

5.2.17. *Проверка орфографии.* Напишите клиент ТТП `SpellChecker`, который принимает в качестве аргумента командной строки имя файла, содержащего словарь английских слов, а затем читает строку из стандартного ввода и выводит все слова, которые отсутствуют в словаре. Используйте множество строк.

5.2.18. *Белый список.* Напишите клиент ТТП, который решает задачу белого списка, первоначально описанную в разделе 1.1 и уточненную в разделе 3.5.

5.2.19. *Случайные телефонные номера.* Напишите клиент `TrieST` (для `R` = 10), который принимает в качестве аргумента командной строки целочисленное значение `N` и выводит `N` случайных телефонных номеров вида `(xxx) xxx-xxxx`. Чтобы не вывести один номер более одного раза, воспользуйтесь таблицей имен. Используйте файл кодов `AreaCodes.txt` с сайта книги, чтобы не выводить несуществующие коды регионов.

5.2.20. *Наличие префикса.* Добавьте в класс `StringSET` (см. упражнение 5.2.6) метод `containsPrefix()`, который принимает в качестве аргумента строку `s` и возвращает `true`, если множество содержит строку с префиксом `s`.

5.2.21. *Сопоставление подстрок.* Пусть задан список (коротких) строк и нужно отвечать на запросы, где пользователь ищет строку `s` — возвращать все строки из списка, которые содержат `s`. Разработайте для этой задачи API и клиент ТТП, реализующий этот API. *Совет*: вставляйте в ТТП суффиксы каждого слова (например, `победа`, `обеда`, `беда`, `еда`, `да`, `а`).

5.2.22. *Печатающие мартышки.* Допустим, что мартышка, печатающая на компьютере, создает случайные слова, добавляя к текущему слову одну из 26 возможных латинских букв с вероятностью p и заканчивая слово с вероятностью $1-26p$. Напишите программу для оценки распределения частот различных длин полученных слов. Если слово появилось несколько раз, считайте его только один раз.

Эксперименты

5.2.23. *Дубликаты (в который раз).* Выполните упражнение 3.5.30, но на основе `StringSET` (см. упражнение 5.2.6), а не `HashSET`. Сравните время выполнения обоих способов. Затем используйте программу `Dedup` для выполнения экспериментов с $N = 10^7$, 10^8 и 10^9, повторите эксперименты для случайных значений `long` и проанализируйте полученные результаты.

5.2.24. *Проверка орфографии.* Выполните упражнение 3.5.31, которое использует файл `dictionary.txt` с сайта книги и клиент `BlackFilter` из листинга 3.5.2 и выводит все ошибочно написанные слова из текстового файла. Сравните производительность реализаций `TrieST` и `TST` для файла `war.txt` с этим клиентом и проанализируйте полученные результаты.

5.2.25. *Словарь.* Выполните упражнение 3.5.32: замеряйте производительность клиента наподобие `LookupCSV` (использующего реализации `TrieST` и `TST`) в ситуации, где важна скорость работы. Точнее, моделируйте ситуацию с генерированием запросов вместо приема команд из стандартного ввода и выполните тесты производительности для больших входных данных и большого количества запросов.

5.2.26. *Индексация.* Выполните упражнение 3.5.33: замеряйте производительность клиента наподобие `LookupIndex` (использующего реализации `TrieST` и `TST`) в ситуации, где важна скорость работы. Точнее, моделируйте ситуацию с генерированием запросов вместо приема команд из стандартного ввода и выполните тесты производительности для больших входных данных и большого количества запросов.

5.3. Поиск подстрок

Фундаментальная операция, которая часто выполняется над строками — *поиск подстроки*: заданы строка *текста* длиной N и строка *образца* длиной M, и нужно найти, в каком месте образец содержится в тексте (рис. 5.3.1). Большинство алгоритмов для этой задачи можно легко расширить для нахождения всех вхождений образца в тексте, для подсчета этих вхождений или для выявления контекста (подстроки текста, окружающие все вхождения образца).

При поиске слова в текстовом редакторе или веб-браузере выполняется как раз поиск подстроки. Первоначально эта задача как раз и возникла в связи с такими поисками. Еще одно классическое приложение — поиск некоторого важного образца в перехваченном обмене сообщениями. Военным интересно найти образец `НАСТУПАЕМ УТРОМ` где-то в перехваченном текстовом сообщении, хакер ищет образец `Password:` в памяти компьютера. В современном мире часто возникает необходимость поиска в большом объеме информации, доступной в веб-сети.

Чтобы лучше представить себе такие алгоритмы, считайте, что образец относительно короткий (M равно, скажем, 100 или 1000), а текст относительно длинный (N равно, скажем, 1 миллиону или 1 миллиарду). При поиске подстрок обычно выполняется предварительная обработка образца, чтобы можно было выполнять быстрые поиски этого образца в тексте.

Поиск подстроки — интересная классическая задача: для ее решения найдено несколько очень непохожих (и неожиданных) алгоритмов, которые не только делают возможным обширный диапазон полезных практических методов, но и демонстрируют несколько фундаментальных приемов построения алгоритмов.

образец → N E E D L E

текст → I N A H A Y S T A C K **N E E D L E** I N A

совпадение

Рис. 5.3.1. *Поиск подстроки*

Краткая история вопроса

Алгоритмы, которые мы рассмотрим, имеют интересную историю; мы приведем ее здесь, чтобы вы имели представление о соотношении различных методов.

Существует примитивный алгоритм поиска подстроки, который довольно широко распространен. В худшем случае время его выполнения пропорционально MN, однако строки, которые встречаются во многих приложениях, обычно приводят к времени выполнения, которое (за исключением патологических случаев) пропорционально $M+N$. Кроме того, он хорошо приспособлен к стандартным архитектурным особенностям большинства компьютерных систем, поэтому его оптимизированная версия показывает такие результаты, которые трудно превзойти даже с помощью хитроумных алгоритмов.

В 1970 г. С. Кук (S. Cook) вывел теоретический результат о некотором виде абстрактной машины — из него следует существование алгоритма, который решает задачу поиска подстроки за время, пропорциональное $M+N$ в худшем случае. Д.Э. Кнут (D.E. Knuth)

и В.Р. Пратт (V.R. Pratt) тщательно проанализировали построение, выполненное Куком для доказательства своей теоремы (которая не предназначалась для практических целей), и преобразовали его в относительно простой и удобный алгоритм. Сначала эта ситуация выглядела как редкий и убедительный пример теоретического результата, который можно было непосредственно (и неожиданно) применить на практике. Однако вскоре оказалось, что Дж.Х. Моррис (J.H. Morris) открыл практически тот же алгоритм, решая неприятную задачу, которая возникла при реализации текстового редактора (он хотел избежать "восстановления" в текстовой строке). Тот факт, что один и тот же алгоритм возник из двух настолько разных подходов, внушает к нему доверие как к фундаментальному решению задачи.

Кнут, Моррис и Пратт не торопились с опубликованием своего алгоритма до 1976 г., а в это время Р.С. Бойер (R.S. Boyer) и Дж.С. Мур (J.S. Moore) — и независимо от них Р.У. Госпер (R.W. Gosper) — открыли алгоритм, который в большинстве случаев работает гораздо быстрее, т.к. он обычно просматривает лишь часть символов в строке текста. Этот алгоритм применяется во многих текстовых редакторах и дает заметное снижение времени отклика при поиске подстрок.

Для работы алгоритмов и Кнута-Морриса-Пратта, и Бойера-Мура необходимо выполнять довольно сложную предварительную обработку образца, которая трудна для понимания и ограничивает сферу их применимости. (Рассказывают даже, что какой-то системный программист счел алгоритм Морриса слишком сложным для понимания и заменил его примитивной реализацией.)

В 1980 г. М.О. Рабин (М.О. Rabin) и Р.М. Карп (R.M. Karp) разработали алгоритм с использованием хеширования, который почти настолько же прост, как примитивный алгоритм, но с очень высокой вероятностью отрабатывает за время, пропорциональное $M+N$. Более того, их алгоритм допускает расширения на двумерный текст и образцы, что делает его полезным для обработки изображений.

Это повествование показывает, что поиск более совершенного алгоритма до сих пор часто увенчивается успехом; и есть подозрение, что на горизонте маячат другие успешные разработки даже для этой классической задачи.

Примитивный поиск подстроки

Очевидный метод поиска подстроки — это проверка во всех позициях текста, где возможно совпадение с образцом, есть ли действительно это совпадение. Метод `search()`, приведенный в листинге 5.3.1 (см. также рис. 5.3.2), как раз выполняет такие действия, чтобы найти первое вхождение строки образца `pat` в троке текста `txt`. В нем используются два указателя: `i` для текста и `j` для образца. Для каждого `i` значение `j` обнуляется, а затем увеличивается до обнаружения или несовпадения, или конца образца (`j == M`). При достижении конца текста (`i == N-M+1`) до обнаружения конца образца фиксируется несовпадение: образец отсутствует в тексте. По нашему соглашению в этом случае возвращается значение `N`, которое означает отсутствие совпадений.

Листинг 5.3.1. Примитивный поиск подстроки

```
public static int search(String pat, String txt)
{
   int M = pat.length();
   int N = txt.length();
```

```
  for (int i = 0; i <= N - M; i++)
  {
     int j;
     for (j = 0; j < M; j++)
        if (txt.charAt(i+j) != pat.charAt(j))
           break;
     if (j == M) return i;     // образец найден
  }
  return N;                    // не найден
}
```

В типичных приложениях обработки текста индекс j увеличивается нечасто, и поэтому время выполнения такого поиска пропорционально N. Почти все сравнения обнаруживают несовпадение уже с первым символом образца. Допустим, мы выполняем поиск образца вхождение в тексте этого абзаца. До конца первого вхождения образца насчитывается 255 символов, среди которых лишь 13 букв в и нет ни одного сочетания вх — поэтому общее количество сравнений символов равно 255 + 13, что дает в среднем 1,051 сравнений на один символ текста. Но нет никаких гарантий, что алгоритм всегда будет работать так эффективно. Например, если образец начинается длинной последовательностью букв A, и текст также содержит длинные последовательности A, то поиск подстроки может замедлиться.

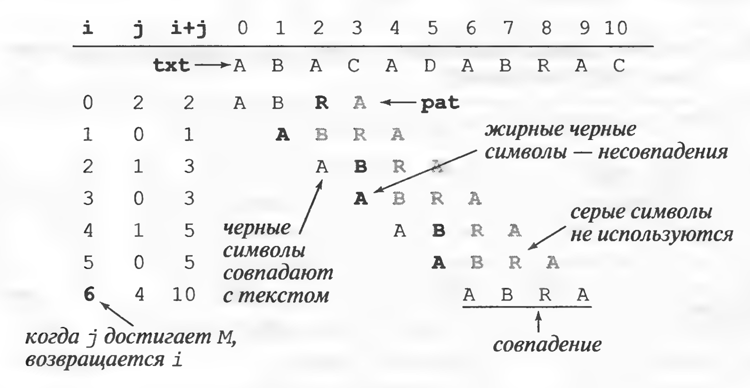


Рис. 5.3.2. Примитивный поиск подстроки

Утверждение Н. При поиске образца длиной M в тексте длиной N примитивный поиск подстроки выполняет в худшем случае $\sim NM$ сравнений символов.

Доказательство. Худший случай входных данных — когда и образец, и текст содержат все символы A, за которыми следует один символ B (рис. 5.3.3). Тогда для каждой из возможных $N-M+1$ позиций сравнения все символы в образце сравниваются с текстом, с общей стоимостью $M(N-M+1)$. Обычно M очень мало по сравнению с N, поэтому среднее значение можно считать равным $\sim NM$.

Такие вырожденные строки вряд ли встретятся в русском тексте, но они вполне вероятны в других приложениях (например, в двоичных файлах), и тогда придется искать лучший алгоритм.

i	j	i+j	0	1	2	3	4	5	6	7	8	9
		txt →	A	A	A	A	A	A	A	A	A	B
0	4	4	A	A	A	A	**B**	← **pat**				
1	4	5		A	A	A	A	**B**				
2	4	6			A	A	A	A	**B**			
3	4	7				A	A	A	A	**B**		
4	4	8					A	A	A	A	**B**	
5	5	10						A	A	A	A	B

Рис. 5.3.3. Примитивный поиск подстроки (худший случай)

Интересна альтернативная реализация, приведенная в листинге 5.3.2. Как и раньше, программа использует один указатель (i) для текста и еще один (j) для образца. Если они указывают на совпадающие символы, увеличиваются оба указателя. Этот код выполняет в точности те же сравнения символов, что и в предыдущей реализации. Чтобы разобраться в его работе, обратите внимание, что переменная i в этом коде содержит значение i+j из предыдущего кода: она указывает на *конец* последовательности уже совпавших символов в тексте (ранее i указывала на *начало* такой последовательности). Если i и j указывают на несовпадающие символы, нужно *восстановить* указатели, чтобы j указывал на начало образца, а i соответствовал сдвигу образца на одну позицию вправо для сравнения с текстом.

Листинг 5.3.2. Альтернативная реализация примитивного поиска подстроки (явное восстановление)

```
public static int search(String pat, String txt)
{
   int j, M = pat.length();
   int i, N = txt.length();
   for (i = 0, j = 0; i < N && j < M; i++)
   {
      if (txt.charAt(i) == pat.charAt(j)) j++;
      else { i -= j; j = 0;  }
   }
   if (j == M) return i - M;    // образец найден
   else        return N;        // не найден
}
```

Алгоритм поиска подстроки Кнута-Морриса-Пратта

Алгоритм, открытый Кнутом, Моррисом и Праттом, основан на следующем принципе: при обнаружении несовпадения мы уже знаем некоторые символы в тексте, т.к. они сравнивались с символами образца до несовпадения. Эта информация позволяет не выполнять восстановление указателя в тексте по всем этим известным символам.

В качестве примера допустим, что используется двухсимвольный алфавит, и выполняется поиск образца BAAAAAAAAA. Пусть было обнаружено совпадение с пятью символами образца, а шестой не совпал. После этого известно, что шесть предыдущих символов в тексте равны BAAAAB (пять совпавших символов и шестой несовпавший), и указатель для текста указывает на последний символ B. Теперь нет необходимости восстанавливать указатель для текста i, т.к. предыдущие четыре символа в тексте содержат A,

т.е. не совпадают с первым символом образца. А вот символ, на который в данный момент указывает i, содержит B и совпадает с первым символом образца, поэтому можно увеличить i и начать со сравнения следующего символа текста со вторым символом образца. Из этого рассуждения следует, что для данного образца можно изменить предложение else в альтернативной примитивной реализации, оставив в нем только j = 1 (без уменьшения i). Поскольку значение i не изменяется внутри цикла, этот метод выполняет не более N сравнений символов. Практический эффект такого изменения ограничен только рассматриваемым образцом (рис. 5.3.4), но сам принцип интересен, и на нем основан алгоритм Кнута-Морриса-Пратта. Любопытно, что *всегда* можно найти такое значение для занесения в указатель j, чтобы не уменьшать указатель i.



Рис. 5.3.4. *Сохранение текстового указателя в поиске подстроки*

Полный пропуск всех совпавших символов при обнаружении несовпадения не работает, если образец совпадает сам с собой с некоторой позиции там, где обнаружено несовпадение. Например, при поиске образца AABAAA в тексте AABAABAAAA вначале обнаруживается несовпадение в позиции 5, но если не откатиться в позицию 3 для продолжения поиска, то совпадение не будет обнаружено. Суть алгоритма Кнута-Морриса-Пратта в том, что можно заранее решить, как продолжать поиск, т.к. это решение зависит только от образца.

Восстановление указателя для образца

В поиске Кнута-Морриса-Пратта подстроки указатель на текст i вообще не восстанавливается, а для записи, насколько нужно откатить указатель j при обнаружении несовпадения, используется массив dfa[][] (рис. 5.3.5). Для каждого символа c элемент dfa[c][j] содержит позицию образца для сравнения со следующей позицией в тексте после сравнения c с pat.charAt(j). Во время выполнения поиска dfa[txt.charAt(i)][j] содержит позицию образца для сравнения с txt.charAt(i+1) после сравнения txt.charAt(i) с pat.charAt(j). При равенстве символов нужно просто перейти к следующему символу, поэтому dfa[pat.charAt(j)][j] всегда равно j+1. А при неравенстве мы знаем не только txt.charAt(i), но и j-1 предыдущих символов в тексте — *это первые* j-1 *символов образца*. Для каждого символа c можно представить, что мы сдвигаем вправо копию образца через эти j символов (первые j-1 из образца с последующим c — т.е. мы решаем, что делать, когда эти символы равны txt.charAt(i-j+1..i)) и останавливаемся при совпадении всех перекрывающихся символов (или обнаруживаем, что таких нет).

j	pat.charAt(j)	dfa[][j] A	B	C	*текст (сам образец)* ABABAC
0	A	1			A
					B
			0		ABABAC
					C
				0	ABABAC
1	B		2		AB
					A**A**
		1			ABABAC
					A**C**
				0	ABABAC
2	A	3			ABA
					AB**B**
			0		ABABAC
					AB**C**
				0	ABABAC
3	B		4		ABAB
					ABA**A**
		1			**A**BABAC
					ABA**C**
				0	ABABAC
4	A	5			ABABA
					ABAB**B**
			0		ABABAC
					ABAB**C**
				0	ABABAC
5	C			6	ABABAC
					ABABA**A**
		1			ABABAC
					ABABA**B**
			4		ABABAC

совпадение (переход к следующему символу); в `dfa[pat.charAt(j)][j]` *заносится j+1*

известный символ текста при несовпадении

несовпадение (откат в образце)

откат выполняется на длину максимального перекрытия начала образца с известными символами текста

Рис. 5.3.5. *Восстановление шаблона для образца* ABABAC *в поиске Кнута-Морриса-Пратта подстроки*

Это дает следующее возможное место, где может быть совпадение с образцом. Индекс символа образца для сравнения с `txt.charAt(i+1)` (`dfa[txt.charAt(i)][j]`) в точности равен количеству перекрывающихся символов.

Метод поиска Кнута-Морриса-Пратта

После вычисления массива `dfa[][]` у нас готов метод поиска подстроки, приведенный в листинге 5.3.3: когда `i` и `j` указывают на различные символы (проверка на совпадение с образцом, начиная с позиции `i-j+1` в строке текста), следующая возможная позиция для сравнения с образцом начинается с позиции `i-dfa[txt.charAt(i)][j]`.

Но по построению первые символы dfa[txt.charAt(i)][j] с этой позиции совпадают с первыми символами dfa[txt.charAt(i)][j], поэтому восстанавливать указатель i нет необходимости: нужно просто занести в j значение dfa[txt.charAt(i)][j] и увеличить i — именно это мы и делаем, когда i и j указывают на одинаковые символы.

Листинг 5.3.3. Поиск Кнута–Морриса–Пратта подстроки (моделирование ДКА)

```
public int search(String txt)
{  // Моделирование работы ДКА с txt.
   int i, j, N = txt.length();
   for (i = 0, j = 0; i < N && j < M; i++)
      j = dfa[txt.charAt(i)][j];
   if (j == M) return i - M;    // образец найден
   else        return N;        // не найден
}
```

Моделирование ДКА

Этот процесс удобно описать в терминах *детерминированного конечного автомата* (ДКА). Вообще-то наш массив dfa[][] как раз и определяет такой автомат. Графическое представление, приведенное на рис. 5.3.6, содержит *состояния* (обозначенные цифрами в кружках) и *переходы* (обозначенные помеченными линиями). Каждому символу образца соответствует одно состояние, и из каждого состояния возможен один переход для каждого символа алфавита. Для рассматриваемых нами ДКА один из переходов — это переход *совпадения* (из j в j+1, помеченный символом pat.charAt(j)), а все остальные — это переходы *несовпадения* (направлены влево).

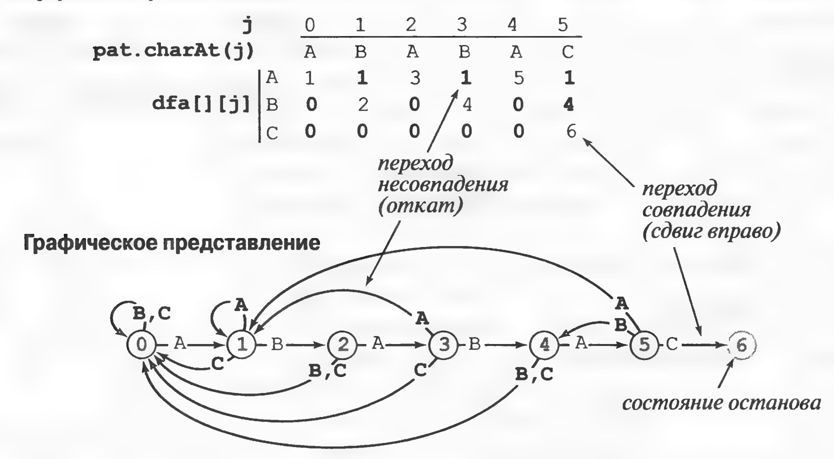


Рис. 5.3.6. ДКА, соответствующий строке ABABAC

Состояния соответствуют сравнениям с символами, по одному для каждого значения индекса образца. Переходы соответствуют изменению значения индекса образца. При рассмотрении i-го символа текста в состоянии j машина выполняет следующее действие: "Выполнить переход в dfa[txt.charAt(i)][j] и сдвинуться на следующий символ (увеличив i)". При переходе совпадения выполняется переход на одну позицию вправо, т.к. dfa[pat.charAt(j)][j] всегда содержит j+1, а при переходе несовпадения выпол-

няется переход влево. Автомат читает символы текста по одному слева направо и каждый раз переходит в новое состояние. Автомат содержит также состояние *останова* M, из которого нет переходов. Машина начинает работу из состояния 0, и если машина достигла состояния M, то это значит, что в тексте найдена подстрока, совпавшая с образцом (мы говорим, что ДКА *распознал* образец), а если машина дошла до конца текста, не достигнув состояния M, то это значит, что образец отсутствует в тексте в виде его подстроки. Каждый образец соответствует некоторому автомату, который представлен массивом переходов dfa[][]. Приведенный в листинге 5.3.3 ДКА-метод поиска подстроки моделирует работу такого автомата.

Чтобы лучше разобраться в работе в ДКА-поиске подстроки, рассмотрим два самых простых действия, которые он выполняет. Когда процесс начинает работу в начале текста в состоянии 0, он остается в состоянии 0, пропуская символы текста, пока не найдет символ, равный первому символу образца, и тогда выполняет переход в следующее состояние. Когда обнаруживается совпадение, выполняется сравнение символов образца с правым концом просмотренного текста с увеличением номера состояния, пока не будет достигнуто состояние M. Трассировка на рис. 5.3.7 содержит типичный пример работы нашего демонстрационного ДКА.

Каждое совпадение сдвигает ДКА в следующее состояние (что эквивалентно увеличению индекса образца j), а каждое несовпадение сдвигает ДКА в предыдущее состояние (что эквивалентно занесению меньшего значения в индекс образца j). Индекс i равномерно проходит по тексту, каждый раз на одну позицию вправо, а индекс j прыгает по образцу так, как указывает ДКА.

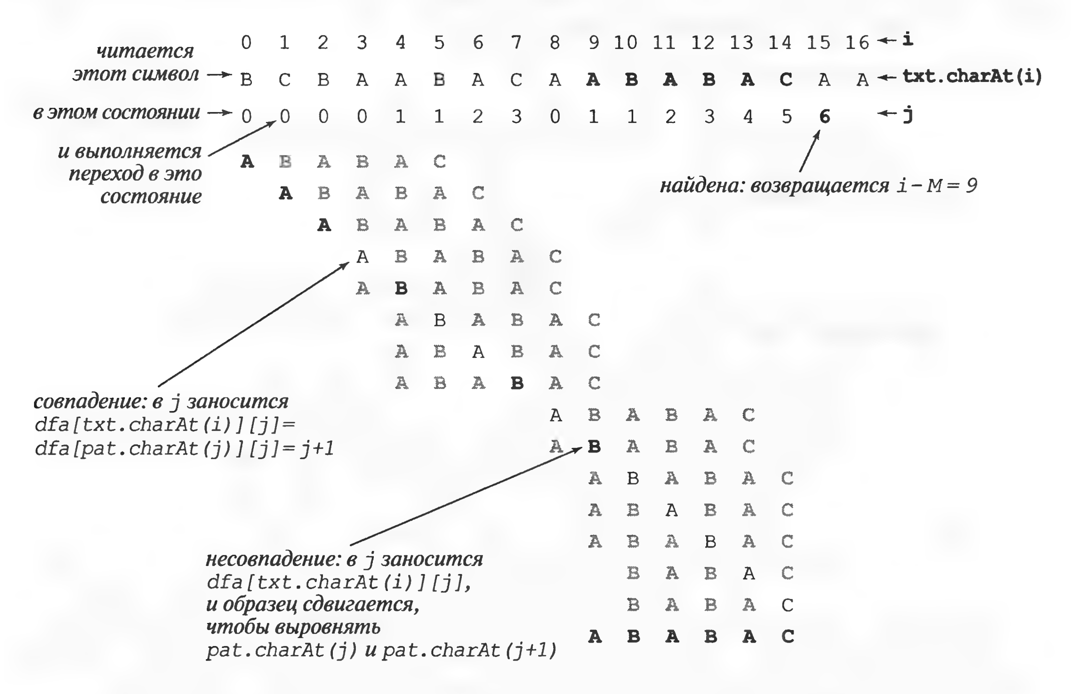


Рис. 5.3.7. Трассировка поиска Кнута-Морриса-Пратта подстроки ABABAC *(моделирование ДКА)*

Создание ДКА

Теперь вы уже понимаете механизм работы, и остался ключевой вопрос, касающийся алгоритма Кнута-Морриса-Пратта — как вычислить массив dfa[][], соответствующий заданному образцу? Любопытно, что ответ на этот вопрос лежит в *самом* ДКА (!) и состоит в хитроумном построении, которое было разработано Кнутом, Моррисом и Праттом. При обнаружении несоответствия в позиции pat.charAt(j) нам нужно знать, в каком состоянии *был бы* ДКА, если восстановить индекс текста и еще раз просмотреть только что просмотренные символы текста после сдвига на одну позицию вправо. Реальное восстановление не нужно, а нужно просто продолжить работу ДКА, *как будто* это восстановление выполнено.

Основное наблюдение состоит в том, что символы в тексте, которые необходимо просмотреть еще раз — это символы с pat.charAt(1) по pat.charAt(j-1): первый символ отбрасывается для сдвига на одну позицию вправо, а последний символ отбрасывается из-за несовпадения. Это уже известные символы образца, поэтому для каждой из возможных позиций несовпадения можно заранее определить состояние, в котором нужно продолжить работу ДКА. На рис. 5.3.8 показаны возможные варианты для нашего примера. *Обязательно разберитесь в этой концепции.*

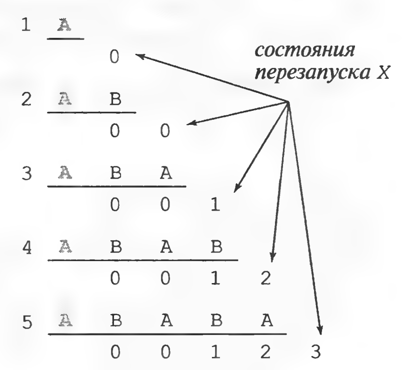


***Рис. 5.3.8.** Моделирование ДКА для вычисления состояний перезапуска для подстроки* ABABAC

Что должен сделать ДКА со следующим символом? Точно то же, что он сделал бы при откате, *за исключением того*, что если он найдет соответствие с pat.charAt(j), то должен перейти в состояние j+1. Например, чтобы решить, что должен сделать ДКА при обнаружении несовпадения в позиции j = 5 подстроки ABABAC, мы с помощью ДКА узнаем, что полный откат приведет нас в состояние 3 для BABA, поэтому можно скопировать dfa[][3] в dfa[][5], а затем занести в элемент для символа C значение 6, т.к. pat.charAt(5) содержит C (совпадение). При построении j-го состояния нужно только знать, как ДКА отрабатывает для j-1 символов, поэтому всю необходимую информацию можно извлечь из частично построенного ДКА.

И последнее важное наблюдение: использование позиции отката X при работе со столбцом j матрицы dfa[][] легко, потому что X < j, и для этого можно использовать частично построенный ДКА — следующее значение X равно dfa[pat.charAt(j)][X].

Для нашего примера из предыдущего абзаца можно заменить значение X на dfa['C'][3] = 0 (но это значение не используется, т.к. построение ДКА завершено).

Вышеприведенное рассуждение приводит к потрясающе компактному коду (см. листинг 5.3.4) для построения ДКА, соответствующего заданному образцу. Для каждого j этот код

- копирует dfa[][X] в dfa[][j] (для случаев несовпадения);
- заносит в dfa[pat.charAt(j)][j] значение j+1 (для случаев совпадения);
- обновляет значение X.

Листинг 5.3.4. Построение ДКА для поиска Кнута–Морриса–Пратта подстроки

```
dfa[pat.charAt(0)][0] = 1;
for (int X = 0, j = 1; j < M; j++)
{  // Вычисление dfa[][j].
   for (int c = 0; c < R; c++)
      dfa[c][j] = dfa[c][X];
   dfa[pat.charAt(j)][j] = j+1;
   X = dfa[pat.charAt(j)][X];
}
```

На рис. 5.3.9 приведена трассировка работы этого кода для нашего примера. Чтобы наверняка понять его, проработайте упражнения 5.3.2 и 5.3.3.

Алгоритм 5.6 в листинге 5.3.5 реализует API, приведенный на рис. 5.3.10.

Листинг 5.3.5. Алгоритм 5.6. Поиск подстроки методом Кнута–Морриса–Пратта

```
public class KMP
{
   private String pat;
   private int[][] dfa;
   public KMP(String pat)
   {  // Построение ДКА для образца.
      this.pat = pat;
      int M = pat.length();
      int R = 256;
      dfa = new int[R][M];
      dfa[pat.charAt(0)][0] = 1;
      for (int X = 0, j = 1; j < M; j++)
      {  // Вычисление dfa[][j].
         for (int c = 0; c < R; c++)
            dfa[c][j] = dfa[c][X];      // Копирование случаев несовпадения.
         dfa[pat.charAt(j)][j] = j+1;   // Оформление случая совпадения.
         X = dfa[pat.charAt(j)][X];     // Обновления состояния перезапуска.
      }
   }
   public int search(String txt)
   {  // Моделирование работы ДКА для txt.
      int i, j, N = txt.length(), M = pat.length();
      for (i = 0, j = 0; i < N && j < M; i++)
         j = dfa[txt.charAt(i)][j];
      if (j == M) return i - M; // образец найден (достигнут конец образца)
      else        return N;     // не найден (достигнут конец текста)
   }
   public static void main(String[] args)
   // См. листинг 5.3.6.
}
```

Конструктор в этой реализации алгоритма Кнута-Морриса-Пратта для поиска подстроки строит ДКА из строки образца, который нужен для выполнения метода `search()` поиска образца в заданной строке текста. Эта программа выполняет те же действия, что и примитивный алгоритм, но работает быстрее для образцов с повторениями.

```
% java KMP AACAA AABRAACADABRAACAADABRA
текст:   AABRAACADABRAACAADABRA
образец:              AACAA
```

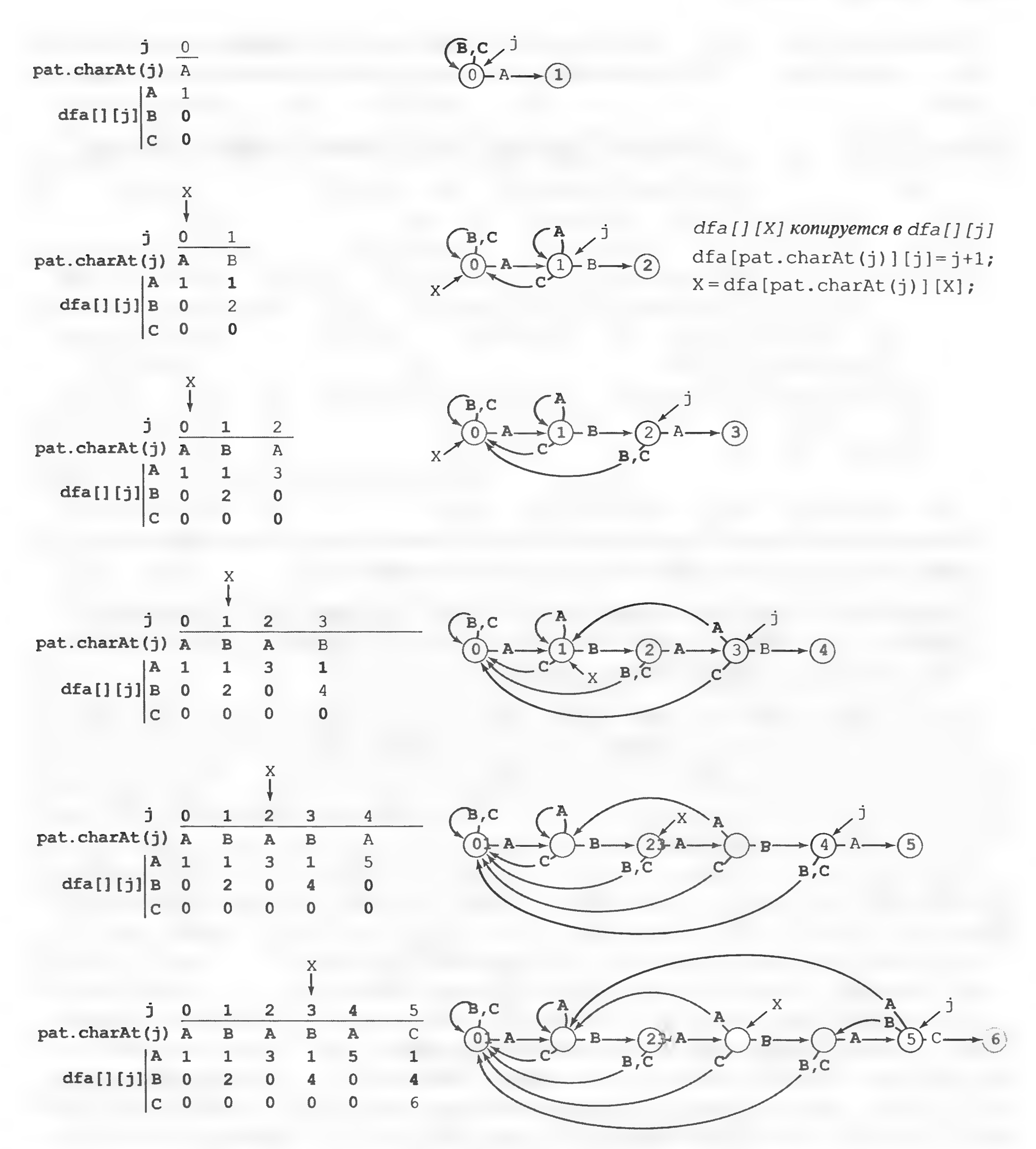


Рис. 5.3.9. Построение ДКА для поиска Кнута-Морриса-Пратта подстроки ABABAC

public class **KMP**		
	KMP(String pat)	*создание ДКА для поиска образца* pat
int	search(String txt)	*поиск индекса* pat *в* txt

Рис. 5.3.10. API поиска подстроки

Типичный клиент тестирования приведен в листинге 5.3.6. Конструктор строит из образца ДКА, который используется в методе search() для поиска образца в заданном тексте.

Листинг 5.3.6. Клиент тестирования для поиска Кнута–Морриса–Пратта подстроки

```
public static void main(String[] args)
{
   String pat = args[0];
   String txt = args[1];
   KMP kmp = new KMP(pat);
   StdOut.println("текст:    " + txt);
   int offset = kmp.search(txt);
   StdOut.print("образец: ");
   for (int i = 0; i < offset; i++)
      StdOut.print(" ");
   StdOut.println(pat);
}
```

Утверждение О. Поиск подстроки длиной M в тексте длиной N методом Кнута-Морриса-Пратта просматривает не более $M+N$ символов.

Доказательство. Непосредственно следует из кода: к каждому символу образца выполняется одно обращение при вычислении dfa[][], и к каждому символу текста тоже один раз (в худшем случае) при работе search().

Можно ввести еще один параметр: для R-символьного алфавита общее время выполнения (и объем памяти), необходимое для построения ДКА, пропорционально MR. Множитель R можно устранить, построив ДКА, в котором каждое состояние содержит переход соответствия и переход несоответствия (а не для каждого возможного символа), но такое построение несколько более сложно.

Линейное время выполнения в худшем случае, гарантированное алгоритмом Кнута-Морриса-Пратта — примечательный теоретический результат. На практике ускорение по сравнению с примитивным методом не всегда заметно, т.к. в весьма немногих приложениях выполняется поиск образцов с многократными повторениями в тексте, который также содержит многократные повторения. Однако этот метод все-таки хорош тем, что он не выполняет откат во входных данных. Это свойство делает поиск Кнута-Морриса-Пратта подстрок более удобным для работы с входным потоком неопределенной длины (вроде стандартного ввода) по сравнению с алгоритмами, которые выполняют восстановление указателя — для чего нужна довольно сложная буферизация. Однако если восстановление выполняется без труда, то такой метод работает значительно лучше, чем метод Кнута-Морриса-Пратта. А теперь мы рассмотрим метод, который обычно приводит к существенному выигрышу в производительности как раз потому, что он *может* выполнять откат в тексте.

Поиск подстроки методом Бойера-Мура

Если откат в строке текста не представляет сложности, можно разработать значительно более быстрый метод поиска подстроки, в котором при сравнении с текстом образец просматривается *справа налево*. Например, если при поиске подстроки BAABBAA обнаружены соответствия на седьмом и шестом символах, но не на пятом, то после

этого образец можно сдвинуть сразу на семь позиций вправо и продолжить проверку с 14-го символа текста, т.к. наше частичное сопоставление обнаружило фрагмент XAA, где X не совпадает с B и не встречается в образце. В общем случае концовка образца может встретиться где угодно, поэтому для позиций продолжения сопоставления нужен массив, как в методе Кнута-Морриса-Пратта. Мы не будем изучать этот подход подробно, т.к. он весьма похож на метод Кнута-Морриса-Пратта. Вместо него мы рассмотрим другой способ, предложенный Бойером (Boyer) и Муром (Moore), который обычное еще более эффективен, чем просмотр образца справа налево.

Как и в реализации поиска Кнута-Морриса-Пратта подстроки, мы решаем, что делать дальше, на основе символа, который привел к несовпадению, в *тексте*, а также в образце. На этапе предобработки нужно решить для каждого символа, который может встретиться в тексте, что следует делать, если этот символ вызовет несовпадение. Простейшая реализация этого принципа приводит к эффективному и полезному методу поиска подстроки.

Эвристика несовпадающего символа

Рассмотрим рис. 5.3.11, где показан поиск образца NEEDLE в тексте FINDINAHAYSTACKNEEDLE. При сопоставлении символов образца справа налево сначала выполняется сравнение самой правой буквы E образца с буквой N текста (в позиции 5). Поскольку буква N присутствует в образце, мы сдвигаем образец на пять позиций вправо, чтобы совместить N в тексте с (самой правой) N в образце. Затем выполняется сравнение самой правой E в образце с буквой S в тексте (в позиции 10). Здесь опять получено несовпадение, но буква S *отсутствует* в образце, и его можно сдвинуть на шесть позиций вправо. Мы обнаруживаем совпадение самой правой E в образце с буквой E в тексте, затем обнаруживаем несовпадающую букву N в позиции 15 и сдвигаем образец вправо на пять позиций, как вначале. И, наконец, перемещаясь влево от позиции 20, мы обнаруживаем, что образец присутствует в тексте. Этот метод находит позицию образца ценой лишь четырех сравнений символов (ну и еще шести для проверки полного совпадения)!

i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
текст →		F	I	N	D	I	**N**	A	H	A	Y	**S**	T	A	C	K	**N**	**E**	**E**	**D**	**L**	**E**	I	N	A
0	5	N	E	E	D	L	**E** ← *образец*																		
5	5						N	E	E	D	L	E													
11	4												N	E	E	D	**L**	**E**							
15	0																**N**	**E**	**E**	**D**	**L**	**E**			

возвращается i=15

Рис. 5.3.11. *Эвристика несовпадающего символа при поиске подстроки справа налево (метод Бойера-Мура)*

Подготовка

Для реализации эвристики несовпадающего символа используется массив right[], который для каждого символа алфавита дает индекс его *самого правого вхождения* в образце (или −1, если такой символ там отсутствует). Это значение указывает, насколько можно сдвинуть индекс, если этот символ обнаружен в тексте и дал несовпадение при поиске подстроки. Для заполнения массива right[] сначала он инициализируется значениями −1, а потом в цикле по j от 0 до M-1 в элемент right[pat.charAt(j)] заносится значение j, как показано на рис. 5.3.12 для нашего демонстрационного образца NEEDLE.

		N	E	E	D	L	E	
c		0	1	2	3	4	5	right[c]
A	-1	-1	-1	-1	-1	-1	-1	-1
B	-1	-1	-1	-1	-1	-1	-1	-1
C	-1	-1	-1	-1	-1	-1	-1	-1
D	-1	-1	-1	-1	**3**	3	3	3
E	-1	-1	**1**	**2**	2	2	**5**	5
...								-1
L	-1	-1	-1	-1	-1	**4**	4	4
M	-1	-1	-1	-1	-1	-1	-1	-1
N	-1	**0**	0	0	0	0	0	0
...								-1

Рис. 5.3.12. Вычисление таблицы пропусков для метода Бойера-Мура

Поиск подстроки

При наличии подготовленного массива right[] реализация алгоритма 5.7 очевидна — см. листинг 5.3.7.

Листинг 5.3.7. Алгоритм 5.7. Поиск подстроки методом Бойера-Мура (эвристика несовпадающего символа)

```
public class BoyerMoore
{
   private int[] right;
   private String pat;

   BoyerMoore(String pat)
   {  // Вычисление таблицы сдвигов.
      this.pat = pat;
      int M = pat.length();
      int R = 256;
      right = new int[R];
      for (int c = 0; c < R; c++)
         right[c] = -1;            // -1 для символов, отсутствующих в образце
      for (int j = 0; j < M; j++)
         right[pat.charAt(j)] = j;  // самая правая позиция для символов в образце
   }

   public int search(String txt)
   {  // Поиск образца в txt.
      int N = txt.length();
      int M = pat.length();
      int skip;
      for (int i = 0; i <= N-M; i += skip)
      {  // Совпадает ли образец с текстом в позиции i ?
         skip = 0;
         for (int j = M-1; j >= 0; j--)
            if (pat.charAt(j) != txt.charAt(i+j))
            {
               skip = j - right[txt.charAt(i+j)];
               if (skip < 1) skip = 1;
               break;
            }
```

```
      if (skip == 0) return i;              // образец найден.
    }
    return N;                               // не найден.
  }
  public static void main(String[] args)    // См. листинг 5.3.6.
}
```

Конструктор в этом алгоритме поиска подстроки строит таблицу, которая содержит для каждого возможного символа самое правое его вхождение в образце. Метод поиска просматривает образец справа налево, выполняя сдвиги для совмещения любого символа, который вызвал несовпадение, с самым правым вхождением в образце.

Индекс `i` перемещается по тексту вправо, а индекс `j` по образцу — влево. Внутренний цикл проверяет, совпадает ли образец с текстом в позиции `i`. Если `txt.charAt(i+j)` равны `pat.charAt(j)` для всех `j` от `M-1` до `0`, то обнаружено совпадение. Иначе имеет место несовпадение и один из трех описанных ниже случаев.

- Если вызвавшего несовпадение символа нет в образце, можно сместить образец на `j+1` позиций вправо (увеличив `i` на `j+1`). Любое меньшее значение совместило бы этот символ с каким-то символом образца. Вообще-то этот перенос совмещает какие-то известные символы в конце образца с его началом, поэтому `i` можно увеличить еще больше, подготовив таблицу вроде Кнута-Морриса-Пратта (рис. 5.3.13).
- Если вызвавший несовпадение символ присутствует в образце, то с помощью массива `right[]` выполняется такое совмещение образца с текстом, чтобы этот символ сравнивался с самым правым вхождением в образце. Для этого `i` увеличивается на `j-right[c]`. Здесь также любое меньшее значение совместило бы этот символ из текста с символом из образца, с которым он точно не совпадает (правее его самого правого вхождения). И здесь также можно усовершенствовать процесс с помощью таблицы наподобие Кнута-Морриса-Пратта, как показано на рис. 5.3.14.
- Если это вычисление не увеличивает индекс `i`, к нему надо прибавить 1, чтобы образец всегда смещался по крайней мере на одну позицию вправо. Нижний пример на рис. 5.3.14 как раз демонстрирует такой случай.

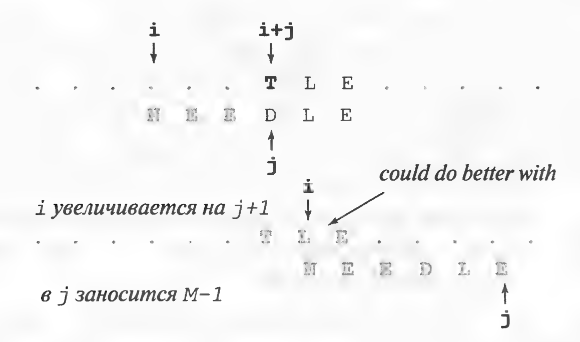


Рис. 5.3.13. Эвристика несовпадающего символа (несовпадение не в образце)

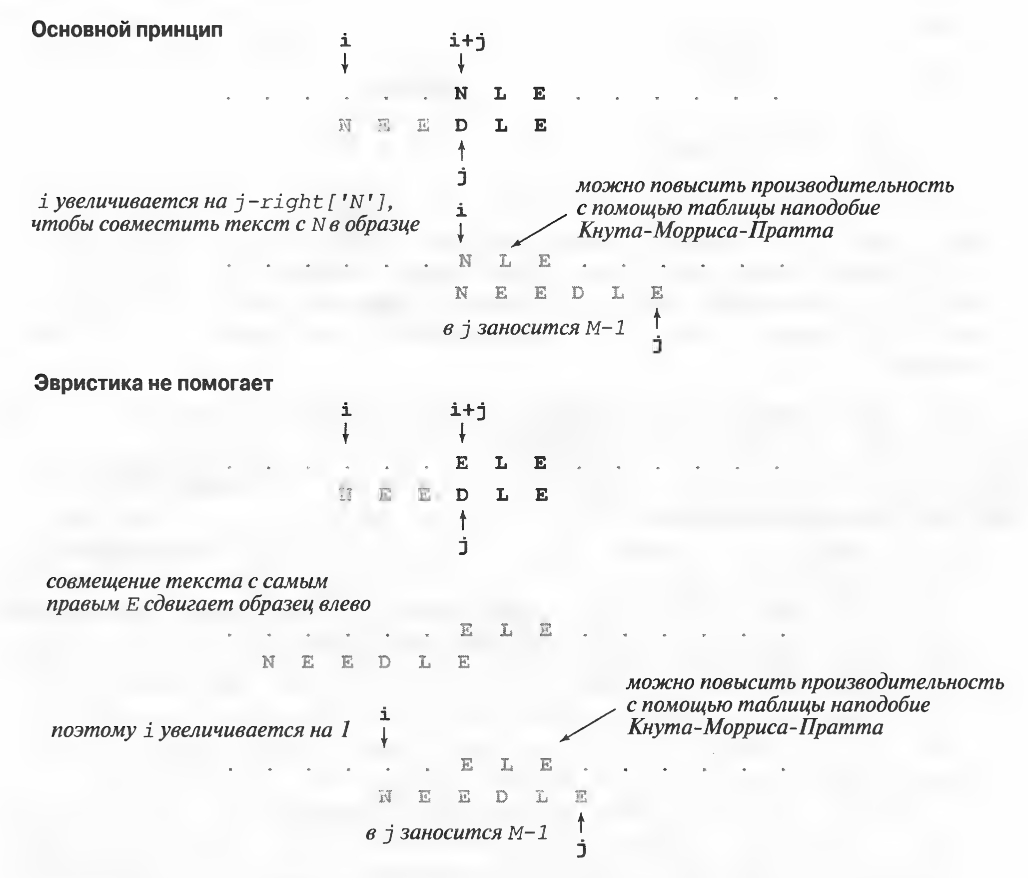


Рис. 5.3.14. *Эвристика несовпадающего символа (несовпадение в образце)*

Алгоритм 5.7 представляет собой естественную реализацию этого процесса. Обратите внимание, что соглашение по использованию −1 в элементах массива `right[]`, соответствующих символам, которые не присутствуют в образце, объединяет первые два случая (`i` увеличивается на `j-right[txt.charAt(i+j)]`).

Полный алгоритм Бойера-Мура учитывает заранее вычисленные несовпадения образца с самим собой (вроде алгоритма Кнута-Морриса-Пратта) и гарантированно отрабатывает за линейное время в худшем случае (а алгоритм 5.7 в худшем случае может потребовать времени, пропорционального NM — см. упражнение 5.3.19). Мы не будем рассматривать эти вычисления, т.к. эвристика несовпадающего символа обеспечивает вполне приемлемую производительность в типичных практических приложениях.

Утверждение П. Для типичных входных данных при выполнении поиска образца длиной M в тексте длиной N метод эвристики несовпадающего символа Бойера-Мура использует $\sim N/M$ сравнений символов.

Обсуждение. Этот результат можно доказать для различных моделей случайных строк, но такие модели не очень похожи на реальные данные, поэтому мы не будем вдаваться в детали. Во многих практических ситуациях бывает так, что в образце отсутствуют почти все символы алфавита, поэтому почти все сравнения приводят к пропуску M символов, что и дает данный результат.

Дактилоскопический поиск Рабина-Карпа

Метод, предложенный М.О. Рабином (М.О. Rabin) и Р.А. Карпом (R.A. Karp), представляет собой совершенно другой подход к поиску подстрок, основанный на хешировании. Для образца вычисляется хеш-функция, а затем ищется совпадение в результате применения такой же хеш-функции к каждой возможной M-символьной подстроке в тексте. Если обнаружено совпадение с такой же хеш-функцией, вычисленной для текстовой подстроки, то можно выполнить проверку на совпадение символов. Этот процесс эквивалентен хранению образца в хеш-таблице, а затем выполнению поиска для каждой подстроки в тексте, но без необходимости выделять память под хеш-таблицу, т.к. достаточно только одного элемента. Непосредственная реализация на основе такого описания работала бы значительно медленнее, чем примитивный поиск (поскольку вычисление хеш-функции, в которую входит каждый символ, требует больше времени, чем просто сравнение символов), однако Рабин и Карп показали, что хеш-функции для M-символьных подстрок несложно вычислять за *константное* время (после выполнения некоторых предварительных действий). Это приводит к поиску подстрок за *линейное* время в практических ситуациях.

Основной принцип

Строка длиной M соответствует M-значному числу по основанию R. Чтобы использовать для таких ключей хеш-таблицу размером Q, необходимо, чтобы хеш-функция преобразовывала M-значные числа по основанию R в целое значение от 0 до $Q-1$. Ответ кроется в модульном хешировании: нужно брать остаток от деления числа на Q. На практике берется произвольное простое число Q, как можно большее, чтобы не возникло переполнение (ведь саму хеш-таблицу хранить не нужно). Этот метод проще понять для небольшого Q и $R = 10$, как на рис. 5.3.15. Чтобы найти образец `26535` в тексте `3141592653589793`, мы выбираем размер таблицы Q (997 в данном примере), вычисляем хеш-значение `26535 % 997 = 613`, а затем ищем соответствие, вычисляя хеш-значения для каждой пятизначной подстроки текста. В данном примере получаются хеш-значения `508`, `201`, `715`, `971`, `442` и `929`, а после них искомое `613`.

```
        pat.charAt(j)
j       0 1 2 3 4
        2 6 5 3 5   % 997 = 613

                          txt.charAt(i)
i       0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
        3 1 4 1 5 9 2 6 5 3 5  8  9  7  9  3
0       3 1 4 1 5   % 997 = 508
1         1 4 1 5 9   % 997 = 201
2           4 1 5 9 2   % 997 = 715
3             1 5 9 2 6   % 997 = 971
4               5 9 2 6 5   % 997 = 442
5                 9 2 6 5 3   % 997 = 929     совпадение
6 <-- возврат i=6   2 6 5 3 5   % 997 = 613 <-
```

Рис. 5.3.15. Принцип поиска подстроки методом Рабина-Карпа

Вычисление хеш-функции

При работе с пятизначными числами можно провести все необходимые вычисления, пользуясь типом int, но что делать, если M равно 100 или 1000? Простое применение метода Горнера, похожего на метод, описанный в разделе 3.4 для строк и других типов ключей с несколькими значениями, приводит к коду, приведенному в листинге 5.3.8. Он вычисляет функцию для M-значного числа по основанию R, представленного в виде массива char, за время, пропорциональное M. (Значение M передается в качестве аргумента, чтобы использовать метод и для образца, и для текста.) Для каждой цифры числа выполняется умножение на R, добавляется эта цифра, и берется остаток от деления на Q. Пример вычисления хеш-функции для нашего образца с помощью этого процесса показан на рис. 5.3.16. Такой же метод может вычислять и хеш-функцию в тексте, но тогда стоимость поиска подстроки будет складываться из умножения, сложения и взятия остатка для каждого символа текста — в худшем случае это NM операций, что не лучше примитивного метода.

Листинг 5.3.8. Метод Горнера для модульного хеширования

```
private long hash(String key, int M)
{  // Вычисление хеша для key[0..M-1].
   long h = 0;
   for (int j = 0; j < M; j++)
      h = (R * h + key.charAt(j)) % Q;
   return h;
}
```

```
                pat.charAt(j)
i      0  1  2  3  4
       2  6  5  3  5
0      2  % 997 = 2          R           Q
1      2  6  % 997 = (2*10 + 6) % 997 = 26
2      2  6  5  % 997 = (26*10 + 5) % 997 = 265
3      2  6  5  3  % 997 = (265*10 + 3) % 997 = 659
4      2  6  5  3  5  % 997 = (659*10 + 5) % 997 = 613
```

Рис. 5.3.16. Вычисление хеш-значения для образца с помощью метода Горнера

Базовый принцип

Метод Рабина-Карпа основан на эффективном вычислении хеш-функции для позиции i+1 текста, если известно ее значение для позиции i. Этот принцип непосредственно следует из простого математического рассуждения. Если обозначить значение txt.charAt(i) как t_i, то число, соответствующее M-символьной подстроке txt, которая начинается в позиции i, равно

$$x_i = t_i R^{M-1} + t_{i+1} R^{M-2} + \ldots + t_{i+M-1} R^0 .$$

Если известно значение $h(x_i) = x_i \bmod Q$, то сдвиг в тексте на одну позицию вправо соответствует замене x_i на

$$x_{i+1} = (x_i - t_i R^{M-1}) R + t_{i+M} .$$

То есть вычитается старшая цифра, выполняется умножение на R, а затем добавляется младшая цифра. Важно то, что нет необходимости хранить значения чисел — достаточно знать значения остатков от их деления на Q. Фундаментальное свойство операции модуля состоит в том, что если после каждой арифметической операции брать остатки от деления на Q, то ответ будет таким же, как после выполнения всех операций с последующим взятием остатка от деления на Q. Мы уже пользовались этим свойством при реализации модульного хеширования методом Горнера (см. листинг 3.4.1). В результате можно эффективно перемещаться по тексту на одну позицию вправо за *константное* время, независимо от того, равно ли M значению 100 или 1000.

Реализация

Это рассуждение непосредственно приводит к реализации поиска подстроки, приведенной в алгоритме 5.8 — см. листинг 5.3.9 и рисунки 5.3.17 и 5.3.18. Конструктор вычисляет хеш-значение образца в `patHash` и значение R^{M-1} mod Q в переменной `RM`. Метод `hashSearch()` начинает работу с вычисления хеш-функции для первых `M` символов текста, а затем сравнивает это значение с хеш-значением образца. Если они не совпали, выполняется проход по строке текста, в котором хеш-функция для `M` символов, начиная с позиции `i`, вычисляется в переменной `txtHash` с помощью вышеописанной техники, и каждое такое значение сравнивается со значением `patHash`. (При вычислении `txtHash` добавляется `Q`, чтобы значение обязательно оставалось положительным для правильного вычисления остатка.)

Листинг 5.3.9. Алгоритм 5.8. Дактилоскопический поиск подстроки методом Рабина–Карпа

```
public class  RabinKarp
{
   private String pat;         // образец (нужен только для варианта Лас-Вегаса)
   private long patHash;       // хеш-значение образца
   private int M;              // длина образца
   private long Q;             // большое простое число
   private int R = 256;        // размер алфавита
   private long RM;            // R^(M-1) % Q

   public RabinKarp(String pat)
   {
      this.pat = pat;          // сохранение образца
                               //   (нужно только для варианта Лас-Вегаса)
      this.M = pat.length();
      Q = longRandomPrime();   // См. упражнение 5.3.33.
      RM = 1;
      for (int i = 1; i <= M-1; i++)      // Вычисление R^(M-1) % Q для
         RM = (R * RM) % Q;               //   удаления старшей цифры.
      patHash = hash(pat, M);
   }

   public boolean check(int i)            // Версия Монте-Карло (см. текст)
   {  return true;  }                     // Для варианта Лас-Вегаса pat
                                          //    сравнивается с txt(i..i-M+1).

   private long hash(String key, int M)
   // См. листинг 5.3.8.
```

```
   private int search(String txt)
   {  // Поиск хеш-значения в тексте.
      int N = txt.length();
      long txtHash = hash(txt, M);
      if (patHash == txtHash) return 0;              // Совпадение с начала.
      for (int i = M; i < N; i++)
      { //Удаление старшей цифры, добавление младшей цифры и проверка на совпадение
         txtHash = (txtHash + Q - RM*txt.charAt(i-M) % Q) % Q;
         txtHash = (txtHash*R + txt.charAt(i)) % Q;
         if (patHash == txtHash)
           if (check(i - M + 1)) return i - M + 1;   // совпадение
      }
      return N;                                      // совпадений не обнаружено
   }
}
```

Этот алгоритм поиска подстроки основан на хешировании. Он вычисляет хеш-значение для образца в конструкторе, а затем выполняет поиск в тексте, ища равенство хешей.

```
                 i   ...  2  3  4  5  6  7  ...
 текущее значение   1  4  1  5  9  2  6  5
    новое значение     4  1  5  9  2  6  5   } текст

                       4  1  5  9  2   текущее значение
                    -  4  0  0  0  0
                          1  5  9  2   вычитание старшей цифры
                             *  1  0   умножение на основание
                       1  5  9  2  0
                                +  6   добавление младшей цифры
                       1  5  9  2  6   новое значение
```

Рис. 5.3.17. Вычисление ключа в поиске подстроки Рабина-Карпа (сдвиг в тексте на одну позицию вправо)

```
 i   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
     3  1  4  1  5  9  2  6  5  3  5  8  9  7  9  3
 0   3  % 997 = 3                        Q
 1   3  1  % 997 = (3*10 + 1) % 997 = 31
 2   3  1  4  % 997 = (31*10 + 4) % 997 = 314
 3   3  1  4  1  % 997 = (314*10 + 1) % 997 = 150
 4   3  1  4  1  5  % 997 = (150*10 + 5) % 997 = 508       RM  R
 5      1  4  1  5  9  % 997 = ((508 + 3*(997 - 30))*10 + 9) % 997 = 201
 6         4  1  5  9  2  % 997 = ((201 + 1*(997 - 30))*10 + 2) % 997 = 715
 7            1  5  9  2  6  % 997 = ((715 + 4*(997 - 30))*10 + 6) % 997 = 971
 8               5  9  2  6  5  % 997 = ((971 + 1*(997 - 30))*10 + 5) % 997 = 442   совпадение
 9                  9  2  6  5  3  % 997 = ((442 + 5*(997 - 30))*10 + 3) % 997 = 929
10 <- возврат          2  6  5  3  5  % 997 = ((929 + 9*(997 - 30))*10 + 5) % 997 = 613
      i-M+6=6
```

Рис. 5.3.18. Пример поиска подстроки методом Рабина-Карпа

Нюанс: статистическая корректность

Когда найдено хеш-значение для M-символьной подстроки `txt`, соответствующее хеш-значению образца, можно ожидать, что код начнет сравнивать эти символы с образцом, чтобы проверить, что обнаружено действительно соответствие, а не коллизия в хеш-таблице. Но мы не выполняем эту проверку, поскольку для нее необходим откат по строке текста. Вместо этого "размер" хеш-таблицы Q берется произвольно большим, т.к. хеш-таблица все равно не создается, и коллизия проверяется лишь с одним ключом — нашим образцом. Мы используем значение `long`, большее 10^{20}, и тогда вероятность, что случайный ключ хешируется в то же значение, что и образец, меньше чем 10^{-20}, а это крайне малая величина. Если она кажется вам не слишком надежной, можно еще раз запустить алгоритм и получить вероятность ошибки меньше 10^{-40}. Этот алгоритм — ранний и знаменитый пример *статистического* алгоритма, который гарантирует время завершения, но с малой вероятностью может выдать неверный ответ. Альтернативный метод проверки на совпадение может работать очень медленно (вплоть до скорости примитивного алгоритма — но тоже с очень малой вероятностью), однако гарантирует верный ответ. Такой алгоритм называется алгоритмом *Лас-Вегаса*.

Утверждение Р. Статистический вариант поиска подстроки методом Рабина-Карпа выполняется за линейное время и почти наверняка выдает правильный результат, а вариант Лас-Вегаса этого поиска точно выдает правильный результат и почти наверняка за линейное время.

Обсуждение. Использование очень большого значения Q (которое возможно в силу того, что сама хеш-таблица не нужна) делает возникновение коллизии крайне маловероятным событием. Рабин и Карп показали, что при правильно выбранном Q коллизия для случайных строк возникает с вероятностью $1/Q$, откуда следует, и для реальных значений переменных это означает, что совпадение хешей не может произойти без совпадения подстрок, а при совпадении подстрок может быть только одно совпадение хешей. Теоретически в какой-то позиции текста может возникнуть коллизия, а не совпадение с подстрокой, но на практике можно ожидать гарантированное совпадение.

Если ваше доверие к теории вероятностей (или к модели случайных строк и нашему коду для генерации случайных чисел) не стопроцентное, вы можете добавить в метод `check()` код проверки, что подстрока текста действительно совпадает с образцом — это превратит алгоритм 5.8 в вариант Лас-Вегаса (см. упражнение 5.3.12). Если еще добавить проверку, что код вообще выполняется, то ваше доверие к теории вероятностей возрастет еще больше.

Поиск подстроки методом Рабина-Карпа называется *дактилоскопическим* поиском, т.к. в нем для представления (потенциально очень большого) образца используется очень небольшой объем информации. Затем этот "отпечаток пальца" (хеш-значение) ищется в тексте. Алгоритм эффективен в силу того, что можно эффективно вычислять и сравнивать "отпечатки" информации.

Резюме

В табл. 5.3.1 приведена сводка сведений по алгоритмам поиска подстрок, с которыми мы познакомились в данном разделе. Как обычно при рассмотрении нескольких алгоритмов для одной и той же задачи, каждый из них имеет свои привлекательные сто-

роны. Примитивный поиск легко реализуется и хорошо работает в типичных случаях (в Java-методе `indexOf()` для типа `String` выполняется примитивный поиск). Алгоритм Кнута-Морриса-Пратта гарантированно выполняется за линейное время без необходимости отката по входным данным, алгоритм Бойера-Мура обычно выполняется за сублинейное время (с коэффициентом M), а алгоритм Рабина-Карпа линеен. Но у каждого есть и недостатки: примитивный алгоритм может потребовать время, пропорциональное MN, алгоритмам Кнута-Морриса-Пратта и Бойера-Мура необходима дополнительная память, а в методе Рабина-Карпа относительно длинный внутренний цикл (несколько арифметических операций вместо сравнений символов в других методах).

Таблица 5.3.1. Сводка стоимостей для реализаций поиска подстроки

Алгоритм	Версия	Количество операций		Откат во входных данных?	Верный результат?	Дополнительная память
		Гарантированное	Типичное			
Примитивный	—	MN	$1{,}1N$	да	да	1
Кнута-Морриса-Пратта	Полный ДКА (алгоритм 5.6)	$2N$	$1{,}1N$	нет	да	MR
	Только переходы при несовпадении	$3N$	$1{,}1N$	нет	да	M
Бойера-Мура	Полный алгоритм	$3N$	N/M	да	да	R
	Только эвристика несовпадающего символа (алгоритм 5.7)	MN	N/M	да	да	R
Рабина-Карпа*	Статистический (алгоритм 5.8)	$7N$	$7N$	нет	да*	1
	Вариант Лас-Вегаса	$7N$*	$7N$	да	да	1

* *вероятностная гарантия для равномерной и независимой хеш-функции*

Вопросы и ответы

Вопрос. **Эта задача поиска подстрок какая-то игрушечная. Что, действительно стоит разбираться в этих сложных алгоритмах?**

Ответ. Ускорение поиска в M раз с помощью метода Бойера-Мура вполне впечатляет. А возможность работы с потоковыми входными данными (без откатов) приводит ко многим практическим приложениям алгоритмов Кнута-Морриса-Пратта и Рабина-Карпа. Кроме таких практических моментов, эта тема предоставляет интересное введение в использование абстрактных машин и рандомизации при проектировании алгоритмов.

Вопрос. **А почему не упростить работу, преобразовав каждый символ в двоичное представление и рассматривая весь текст как двоичный?**

Ответ. Эта идея не очень-то эффективна из-за возможности ложных совпадений на границах символов.

Упражнения

5.3.1. Напишите примитивную реализацию поиска подстроки Brute, используя тот же API, что и алгоритм 5.6.

5.3.2. Приведите содержимое массива dfa[][] для образца AAAAAAAAA для алгоритма Кнута-Морриса-Пратта и нарисуйте ДКА в стиле рисунков в тексте.

5.3.3. Приведите содержимое массива dfa[][] для образца ABRACADABRA для алгоритма Кнута-Морриса-Пратта и нарисуйте ДКА в стиле рисунков в тексте.

5.3.4. Напишите эффективный метод, который принимает в качестве аргументов строку txt и целое число M и возвращает позицию первого вхождения M последовательных пробелов в строке или txt.length, если такого вхождения нет. Оцените количество сравнений символов, которые выполняются в вашем методе, для типичного текста и в худшем случае.

5.3.5. Разработайте примитивную реализацию поиска подстроки BruteForceRL, которая просматривает образец справа налево (упрощенный вариант алгоритма 5.7).

5.3.6. Приведите содержимое массива right[], вычисленного конструктором в алгоритме 5.7 для образца ABRACADABRA.

5.3.7. Добавьте в нашу примитивную реализацию поиска подстроки метод count(), который подсчитывает вхождения подстроки, и метод searchAll(), который выводит все вхождения.

5.3.8. Добавьте в класс KMP метод count(), который подсчитывает вхождения подстроки, и метод searchAll(), который выводит все вхождения.

5.3.9. Добавьте в класс BoyerMoore метод count(), который подсчитывает вхождения подстроки, и метод searchAll(), который выводит все вхождения.

5.3.10. Добавьте в класс RabinKarp метод count(), который подсчитывает вхождения подстроки, и метод searchAll(), который выводит все вхождения.

5.3.11. Придумайте пример худшего случая для реализации Бойера-Мура в алгоритме 5.7 (который демонстрирует выполнение за более чем линейное время).

5.3.12. Добавьте в класс RabinKarp (алгоритм 5.8) метод check(), который превращает его в алгоритм Лас-Вегаса (проверяйте, что образец совпадает с текстом в позиции, заданной аргументом).

5.3.13. Покажите, что в реализации метода Бойера-Мура в алгоритме 5.7 можно занести в right[c] предпоследнее вхождение символа c, если он является последним символом в образце.

5.3.14. Разработайте варианты реализаций поиска подстроки из данного раздела, в которых для представления образца и текста применяется массив char[], а не тип String.

5.3.15. Предложите примитивный алгоритм поиска подстроки, который просматривает образец справа налево.

5.3.16. Приведите трассировку работы примитивного алгоритма в стиле приведенных в тексте рисунков для следующих строк образца и текста:

а) *образец*: AAAAAAAB *текст*: AAAAAAAAAAAAAAAAAAAAAAAAB

б) *образец*: ABABABAB *текст*: ABABABABAABABABABAAAAAAAA

5.3.17. Нарисуйте ДКА метода Кнута-Морриса-Пратта для следующих строк образца:

а) AAAAAAB

б) AACAAAB

в) ABABABAB

г) ABAABAAABAAAB

д) ABAABCABAABCB

5.3.18. Пусть образец и текст — *случайные* строки из алфавита размером R (не менее 2). Покажите, что ожидаемое количество сравнений символов для примитивного алгоритма равно $(N - M + 1)\,(1 - R^{-M})\;/\;(1 - R^{-1}) \le 2(N - M + 1)$.

5.3.19. Придумайте пример, для которого алгоритм Бойера-Мура (только с эвристикой несовпадающего символа) имеет плохую производительность.

5.3.20. Как следует изменить алгоритм Рабина-Карпа, чтобы определить, присутствует ли в тексте любой из множества k образцов (одинаковой длины)?

Решение: вычислите хеши для всех k образцов и храните их в объекте `StringSET` (см. упражнение 5.2.6).

5.3.21. Как следует изменить алгоритм Рабина-Карпа, чтобы выполнять поиск заданного образца с дополнительным условием, что средний символ является “обобщенным” (т.е. любым символом, который может встретиться в тексте)?

5.3.22. Как следует изменить алгоритм Рабина-Карпа, чтобы выполнять поиск образца размером $H \times M$ в тексте размером $N \times N$?

5.3.23. Напишите программу, которая считывает символы по одному и сообщает о каждом случае, когда текущая строка является палиндромом. *Совет*: используйте принцип хеширования Рабина-Карпа.

Творческие задачи

5.3.24. *Поиск всех вхождений.* Добавьте во все четыре алгоритма поиска подстроки, приведенные в тексте, метод `findAll()`, который возвращает объект `Iterable<Integer>`, позволяющий клиентам перебрать все позиции, где образец встречается в тексте.

5.3.25. *Потоковый ввод.* Добавьте в класс `KMP` метод `search()`, который принимает в качестве аргумента переменную типа `In` и выполняет поиск образца в указанном входном потоке *без* использования дополнительных переменных экземпляров. Затем сделайте то же самое с классом `RabinKarp`.

5.3.26. *Проверка наличия циклических перестановок.* Напишите программу, которая для заданных двух строк определяет, является ли одна из них циклической перестановкой другой — например, `пальто` и `топаль`.

5.3.27. *Поиск кратного повторения.* Кратным повторением базовой строки `b` в строке `s` является подстрока из `s`, содержащая по крайней мере две последовательные копии `b` (без перекрытия). Разработайте и реализуйте алгоритм с линейным временем выполнения, который для двух заданных строк `b` и `s` возвращает индекс начала самого длинного кратного повторения `b` в `s`.

Например, для b = abcab и s = abcabcababcababcababcac такая программа должна возвратить результат 3.

5.3.28. *Буферизация в примитивном алгоритме.* Добавьте в решение упражнения 5.3.1 метод search(), который принимает в качестве аргумента входной поток (типа In) и выполняет в нем поиск образца. *Примечание*: придется задействовать буфер, который может хранить, по крайней мере, M предыдущих символов из входного потока. Основная трудность здесь — написать эффективный код для инициализации, обновления и очистки буфера для любого входного потока.

5.3.29. *Буферизация в методе Бойера-Мура.* Добавьте в алгоритм 5.7 метод search(), который принимает в качестве аргумента входной поток (типа In) и выполняет в нем поиск образца.

5.3.30. *Двумерный поиск.* Реализуйте вариант алгоритма Рабина-Карпа для поиска образцов в двумерном тексте. Образец и текст представляют собой прямоугольные блоки символов.

5.3.31. *Случайные образцы.* Сколько сравнений символов требуется для поиска подстроки, соответствующей случайному образцу длиной 100 в заданном тексте?

Ответ: ни одного. Метод

```
public boolean search(char[] txt)
{  return false; }
```

вполне эффективно решает эту задачу, т.к. вероятность появления случайного образца длиной 100 в любом тексте настолько мала, что ее можно считать нулевой.

5.3.32. *Уникальные подстроки.* Выполните упражнение 5.2.14 с помощью принципа, на котором основан метод Рабина-Карпа.

5.3.33. *Случайные простые числа.* Реализуйте метод longRandomPrime() для класса RabinKarp (алгоритм 5.8). *Подсказка*: случайное n-значное число является простым с вероятностью, пропорциональной $1/n$.

5.3.34. *Линейный код.* Виртуальная машина Java (и язык ассемблера вашего компьютера) поддерживают инструкцию goto, которая позволяет приблизить программу к машинному коду — наподобие программы в листинге 5.3.10, которая в точности моделирует ДКА для этого образца, но более эффективна. Чтобы не выполнять проверку на достижение конца текста при каждом увеличении i, мы считаем, что этот образец находится в тексте и отмечает его конец. Метки в операторах goto соответствуют массиву dfa[]. Напишите статический метод, который принимает образец в качестве входного параметра и выдает линейную программу для поиска этого образца.

Листинг 5.3.10. Линейный поиск подстроки AABAAA

```
    int i = -1;
sm: i++;
s0: if (txt[i]) != 'A' goto sm;
s1: if (txt[i]) != 'A' goto s0;
s2: if (txt[i]) != 'B' goto s0;
s3: if (txt[i]) != 'A' goto s2;
s4: if (txt[i]) != 'A' goto s0;
s5: if (txt[i]) != 'A' goto s3;
    return i-8;
```

23 Зак. 3239

5.3.35. *Поиск Бойера-Мура в двоичных строках.* Эвристика несовпадающего символа не особенно годится для двоичных строк, т.к. есть лишь два возможных символа, которые могут привести к несоответствию, и они присутствуют в образце с примерно одинаковой вероятностью. Разработайте класс поиска подстроки для двоичных строк и групп битов, в котором используются "символы" из нескольких битов, в точности как в алгоритме 5.7. *Примечание*: если одновременно выбирать по b битов, то понадобится массив `right[]` с 2^b элементами. Значение b следует выбрать достаточно небольшим, чтобы эта таблица не получилась огромной, но все-таки достаточно большим, чтобы большинство b-битовых фрагментов в тексте не часто встречались в образце: в образце может быть $M - b + 1$ различных b-битовых фрагментов (по одному, начиная с каждой битовой позиции от 1 до $M - b + 1$), поэтому нужно, чтобы $M - b + 1$ было значительно меньше 2^b. Например, если взять 2^b равным примерно $\lg(4M)$, то массив `right[]` будет более чем на три четверти заполнен элементами –1, но b нельзя опустить ниже $M/2$, поскольку иначе можно полностью пропустить образец, если он окажется разбитым между двумя b-битовыми фрагментами текста.

Эксперименты

5.3.36. *Случайный текст.* Напишите программу, которая принимает в качестве аргументов целые числа `M` и `N`, генерирует случайную строку двоичного текста длиной `N`, а затем подсчитывает количество вхождений последних `M` битов в других местах этой строки. *Примечание*: для различных значений `M` могут быть удобны различные методы.

5.3.37. *Поиск Кнута-Морриса-Пратта для случайного текста.* Напишите клиент, который принимает в качестве входных параметров целые значения `M`, `N` и `T` и выполняет `T` раз следующий эксперимент. Генерируется случайный образец длиной `M` и случайный текст длиной `N`, а затем подсчитывается количество сравнений символов при поиске образца в тексте с помощью реализации `KMP`. Добавьте в класс `KMP` возможность подсчета сравнений и выведите среднее значение для `T` повторений.

5.3.38. *Поиск Бойера-Мура для случайного текста.* Выполните предыдущее упражнение для класса `BoyerMoore`.

5.3.39. *Замеры времени.* Напишите программу, которая выполняет замеры времени выполнения для поиска всеми четырьмя методами подстроки

```
it is a far far better thing that i do than i have ever done
```

в тексте "Повести о двух городах" (`tale.txt`). Объясните степень совпадения, до которой ваши результаты подтверждают сведения о производительности, приведенные в тексте раздела.

5.4. Регулярные выражения

Во многих приложениях бывает необходимо выполнить поиск подстроки на основании не очень полной информации об искомом образце. Пользователю текстового редактора может понадобиться указать только часть образца или образец, который может соответствовать нескольким различным словам, либо указать, что допустимо совпадение с любым из нескольких заданных образцов. Биологу может понадобиться найти последовательность геномов, удовлетворяющую некоторым условиям. В данном разделе мы узнаем, как можно эффективно выполнить такое сопоставление.

Алгоритмы из предыдущего раздела по своей сути зависят от точно заданного образца, поэтому нам понадобятся другие методы. Базовые механизмы, которые мы рассмотрим, позволяют выполнять очень мощные поиски сложных M-символьных образцов в N-символьных строках текста за время, в худшем случае пропорциональное MN, а в типичных случаях намного быстрее.

Для начала нам нужен способ описания образцов — четкий способ для указания различных вышеописанных задач поиска частичного совпадения. В этом описании должны содержаться более мощные примитивные операции, чем операция "проверить, совпадает ли i-й символ из строки текста с j-м символом образца", которой вполне хватало в предыдущем разделе. Для этого традиционно используются *регулярные выражения*, которые описывают образцы с помощью трех естественных, простых, но мощных операций.

Программисты пользуются регулярными выражениями уже не один десяток лет. Взрывообразный рост возможностей поиск в веб-сети сделал их еще более распространенными. В начале этого раздела мы рассмотрим ряд конкретных приложений, чтобы не только дать вам почувствовать их пользу и мощь, но и ознакомить вас с их базовыми свойствами.

Как и в случае алгоритма Кнута-Морриса-Пратта из предыдущего раздела, мы рассмотрим эти три простейшие операции в терминах абстрактной машины, которая может выполнять поиск образцов в строке текста. Затем, как и раньше, наш алгоритм сопоставления с образцом будет строить такие машины и моделировать их работу. Конечно, машины сопоставления с образцом обычно более сложны, чем детерминированный конечный автомат (ДКА) для метода Кнута-Морриса-Пратта, но не так уж сложны, как можно было подумать.

Вы увидите, что решение, которое мы разработаем для задачи сопоставления с образцом, тесно связано с фундаментальными процессами в компьютерных науках. Например, метод, которым мы воспользуемся в нашей программе для выполнения поиска в строке на основе заданного описания образца, аналогичен методу, который применяется в системе Java для преобразования конкретной Java-программы в программу на машинном языке для вашего компьютера. Мы также рассмотрим концепцию *недетерминизма*, который играет важную роль в эффективных алгоритмах поиска (см. главу 6).

Описание образцов с помощью регулярных выражений

Мы рассмотрим описания образцов с помощью символов, которые служат операндами для трех фундаментальных операций. В этом контексте мы будем использовать слово *язык* в особом смысле — для обозначения множества строк (возможно, бесконечного), а слово *образец* — для обозначения описания языка. Правила, которые мы будем рассматривать, весьма похожи на аналогичные правила для описания арифметических выражений.

Конкатенация

Первая фундаментальная операция уже использовалась в предыдущем разделе. Запись AB означает, что определяется язык {AB}, содержащий одну двухбуквенную строку, которая получена слитной записью A и B.

Или

Вторая фундаментальная операция позволяет указывать альтернативы в образце. Если операция *или* находится между двумя альтернативными вариантами, то они оба присутствуют в языке. Для обозначения этой операции мы будем использовать вертикальную черточку |. Например, выражение A|B задает язык {A, B}, а выражение A|E|I|O|U задает язык {A, E, I, O, U}. Конкатенация имеет более высокий приоритет, чем *или*, поэтому AB|BCD задает язык {AB, BCD}.

Замыкание

Третья фундаментальная операция позволяет повторять части образца произвольное число раз. *Замыкание* образца — это язык строк, сформированных конкатенацией образца с самим собой любое (в том числе и нулевое) количество раз. Для обозначения этой операции мы будем использовать звездочку * после повторяемого образца. Замыкание имеет более высокий приоритет по сравнению с конкатенацией, поэтому выражение AB* задает язык, который состоит из строк из буквы A, за которой находятся 0 или более B, а A*B задает язык, который состоит из строк из 0 или более A, за которой находится одна B. *Пустая строка*, которую мы будем обозначать ϵ, присутствует в каждой текстовой строке (и в A* тоже).

Скобки

Скобки применяются для перекрытия стандартных правил предшествования. К примеру, выражение C(AC|B)D задает язык {CACD, CBD}; (A|C)((B|C)D) задает язык {ABD, CBD, ACD, CCD}; а (AB)* задает язык из строк, сформированных из любого количества повторений AB, в том числе и без повторений: {ϵ, AB, ABAB, ...}.

Эти простые правила позволяют записывать регулярные выражения, которые могут выглядеть сложными, но они четко и полно описывают языки (см. несколько примеров в табл. 5.4.1). Часто язык можно описать и каким-то другим способом, но найти такое описание не всегда просто. Например, регулярное выражение в нижней строке табл. 5.4.1 задает подмножество языка (A|B)* с четным количеством B.

Регулярные выражения представляют собой очень простые формальные объекты — они даже проще арифметических выражений, с которыми вы познакомились еще в средней школе. И эту простоту можно использовать для разработки компактных и эффективных алгоритмов для их обработки. Начнем мы со следующего формального определения.

Таблица 5.4.1. Примеры регулярных выражений

Регулярное выражение	Соответствует	Не соответствует
(A|B)(C|D)	AC AD BC BD	любой другой строке
A(B|C)*D	AD ABD ACD ABCCBD	BCD ADD ABCBC
A* | (A*BA*BA*)*	AAA BBAABB BABAAA	ABA BBB BABBAAA

Определение. Регулярное выражение (РВ) — это один из вариантов:

- пусто;
- один символ;
- регулярное выражение, заключенное в скобки;
- два или более *конкатенированных* регулярных выражения;
- два или более регулярных выражения, разделенных операцией *или* (|);
- регулярное выражение с последующей операцией *замыкания* (*).

Это определение описывает *синтаксис* регулярных выражений и сообщает, из чего состоят допустимые регулярные выражения. *Семантика*, которая поясняет значение данного регулярного выражения, определяется неформальными описаниями, которые приведены в данном разделе. Для справки мы приведем их в компактном виде, продолжив запись формального определения.

Определение (продолжение). Каждое РВ представляет множество строк, определенных следующим образом.

- Пустое РВ представляет *пустое* множество строк, содержащее 0 элементов.
- Один символ представляет множество строк из одного элемента — этого символа.
- РВ в скобках представляет то же множество строк, что и РВ без скобок.
- РВ, состоящее из двух *конкатенированных* РВ, представляет *прямое произведение* множеств строк, представленных отдельными компонентами (все возможные строки, которые можно сформировать, взяв по одной строке из каждого компонента и записав слитно в том же порядке, что и РВ-компоненты).
- РВ, состоящее из двух РВ с операцией *или* между ними, представляет *объединение* множеств, представленных отдельными компонентами.
- РВ, состоящее из *замыкания* РВ, представляет ϵ (пустую строку) или объединение множеств, представленных конкатенацией любого количества копий этого РВ.

В общем случае язык, описанный некоторым РВ, может быть очень большим и даже бесконечно большим. Каждый язык можно описать многими различными способами, и необходимо опробовать различные образцы, точно так же, как мы пробуем писать компактные программы и реализовывать эффективные алгоритмы.

Сокращения

В типичных приложениях принимаются различные дополнения к этим базовым правилам, которые позволяют разрабатывать сжатые описания нужных языков. С теоретической точки зрения такие дополнения представляют собой просто сокращения для последовательности операций со многими операндами; а с практической точки зрения это очень полезные расширения базового набора операций, которые позволяют записывать компактные образцы.

Описатели множеств символов

Часто бывает удобно задать множество символов одним символом или короткой последовательностью (табл. 5.4.2). Символ точки (.) является *обобщенным символом* (wildcard), который представляет любой одиночный символ. Последовательность символов в квадратных скобках представляет любой один из этих символов. Последовательность можно также задать как диапазон символов. Если перед последовательностью находится символ ^, то последовательность в квадратных скобках представляет любой символ, *кроме* указанных. Эти обозначения являются просто сокращениями для последовательностей операций *или*.

Таблица 5.4.2. Описатели множеств символов

Название	Обозначение	Пример
Обобщенный символ	.	A.B
Один из символов	Символы в []	[AEIOU]*
Диапазон	Символы в [], разделенные дефисом	[A-Z], [0-9]
Дополнение	Символы в [], а перед ними ^	[^AEIOU]*

Сокращения для замыканий

Операция замыкания задает любое количество копий своего операнда. На практике часто бывает нужно указать конкретное число копий или диапазон такого числа. В частности, знак + означает как минимум одну копию, знак ? — ноль или одну копию, а счетчик или диапазон в фигурных скобках ({}) — заданное количество копий. Эти обозначения также представляют собой сокращения для последовательностей базовый операций конкатенации, *или* и замыкания.

Таблица 5.4.3. Сокращения для замыканий (указание количества копий операнда)

Вариант	Обозначение	Пример	Сокращение для	В языке	Не в языке
Хотя бы 1	+	(AB)+	(AB)(AB)*	AB ABABAB	ϵ BBBAAA
0 или 1	?	(AB)?	ϵ | AB	ϵ AB	любая другая строка
Конкретно	Счетчик в {}	(AB){3}	(AB)(AB)(AB)	ABABAB	любая другая строка
Диапазон	Диапазон в {}	(AB){1-2}	(AB)|(AB)(AB)	AB ABAB	любая другая строка

Управляющие последовательности

Некоторые символы — \, ., |, *, (и) — представляют собой *метасимволы*, которые используются для построения регулярных выражений. И чтобы представить в образцах эти символы алфавита, используются *управляющие последовательности*, которые начинаются символом обратной косой черты \. Управляющая последовательность — это символ \ с последующим одним метасимволом (который представляет этот самый символ). Например, последовательность \\ представляет символ \. Другие управляющие последовательности представляют другие специальные и пробельные символы. Например, \t представляет символ табуляции, \n — символ новой строки, а \s — любой пробельный символ.

Регулярные выражения в приложениях

РВ оказались очень удобными для описания языков, которые нужны в практических приложениях. Поэтому они интенсивно используются и интенсивно изучаются. Чтобы плотнее познакомиться с регулярными выражениями, а также почувствовать их пользу, мы, прежде чем приступить к изучению алгоритма сопоставления с РВ-образцами, рассмотрим ряд практических приложений (табл. 5.4.4). РВ также играют важную роль в теории компьютерных наук. Полное обсуждение этой роли выходит за рамки данной книги, но иногда мы будем упоминать соответствующие фундаментальные результаты.

Таблица 5.4.4. Типичные регулярные выражения в приложениях (упрощенные)

Контекст	Регулярное выражение	Соответствует
Поиск подстрок	`.*NEEDLE.*`	`A HAYSTACK NEEDLE IN`
Номер телефона	`\([0-9]{3}\)\[0-9]{3}-[0-9]{4}`	`(800) 867-5309`
Идентификатор Java	`[$_A-Za-z][$_A-Za-z0-9]*`	`Pattern_Matcher`
Маркер генома	`gcg(cgg|agg)*ctg`	`gcgaggaggcggcggctg`
Адрес электронной почты	`[a-z]+@([a-z]+\.)+(edu|com)`	`rs@cs.princeton.edu`

Поиск подстрок

Наша основная цель — разработать алгоритм, который определяет, входит ли заданная строка текста во множество строк, описанных заданным регулярным выражением. Если текст принадлежит языку, описанному образцом, мы говорим, что текст *соответствует* образцу. Сопоставление с образцами, заданными с помощью РВ, в значительной мере обобщает задачу поиска подстроки из раздела 5.3. И для поиска подстроки pat в строке текста txt нужно проверить, принадлежит ли txt языку, описываемому образцом .*pat.*, или нет.

Проверка допустимости

Сравнение с РВ часто выполняется при работе с веб-сетью. После ввода даты или номера счета программа обработки входных данных должна проверить, ввели ли вы данные в правильном формате. Можно написать код, который проверяет все возможные случаи: если вы должны ввести сумму в долларах, код может проверить, ввели ли вы перед суммой символ $, а сама сумма записывается набором десятичных цифр и т.д.

Но лучше определить РВ, которое описывает все допустимые входные данные. Тогда проверка допустимости входных данных сводится к задаче сопоставления с образцом: принадлежат ли введенные данные языку, который описан заданным РВ? Библиотеки различных регулярных выражений для часто выполняемых проверок плодятся в Интернете, т.к. этот вид проверки очень распространен. Как правило, РВ представляет собой гораздо более точное и сжатое описание множества всех доступных строк, чем программа, проверяющая все возможные случаи.

Инструментальный набор программиста

Прародитель сопоставления с регулярными выражениями — Unix-команда `grep`, которая выводит все строки, соответствующие заданному РВ. Эта утилита зарекомендовала себя бесценным средством для поколений программистов, и теперь РВ применяются во многих современных системах программирования, от `awk` и `emacs` до Perl, Python и JavaScript. Например, предположим, что имеется каталог со многими десятками `.java`-файлов, и нужно найти тот, код в котором использует `StdIn`. Команда

```
% grep StdIn *.java
```

сразу же даст ответ. Она выведет все строки, соответствующие `.*StdIn.*` в каждом файле.

Геномика

Биологи используют РВ для решения важных научных задач. Например, последовательность генов человека содержит область, которую можно описать регулярным выражением `gcg(cgg)*ctg`, где количество повторений фрагмента `cgg` сильно отличается у разных людей, а с большим количеством повторений связано определенное генетическое заболевание, которое приводит к задержке умственного развития и другим симптомам.

Поиск

Поисковые механизмы в веб-сети поддерживают РВ, хотя и не в полном объеме. Обычно можно указывать альтернативы (`|`) и повторения (`*`).

Возможности

При первом знакомстве с теоретическими компьютерными науками часто рассматривается множество языков, которые можно задать с помощью РВ. Вы можете удивиться, узнав, что операцию модуля можно реализовать с помощью РВ; например, выражение `(0 | 1(01*0)*1)*` описывает все строки нулей и единиц, представляющие двоичные числа, кратные трем (!): строки `11`, `110`, `1001` и `1100` принадлежат языку, а `10`, `1011` и `10000` — нет.

Ограничения

Не все языки можно задать с помощью регулярных выражений. Забавным примером является то, что ни одно РВ не может описать множество строк, которые задают все допустимые РВ. Более простые примеры, основанные на том же принципе — РВ не позволяют проверить сбалансированность скобок или проверить, что строка содержит равное количество букв `A` и `B`.

Недетерминированные конечные автоматы

Вспомните, что алгоритм Кнута-Морриса-Пратта можно рассматривать как конечную машину для просмотра текста, построенную из искомого образца. Для сопоставления с регулярными выражениям эту идею можно расширить.

Конечный автомат для алгоритма Кнута-Морриса-Пратта переходит из состояния в состояние в зависимости от просматриваемых символов из строки текста. Такой автомат сообщает о соответствии в том и только в том случае, если он достигает своего конечного состояния. Сам алгоритм сводится к моделированию работы автомата. Облегчает такое моделирование *детерминированность* автомата: каждый переход в новое состояние полностью определяется следующим символом текста.

Для обработки регулярных выражений мы рассмотрим более мощную абстрактную машину. В силу наличия операции *или* автомат не может определить, возможно ли совпадение с образцом, на основании анализа только одного символа. Вообще-то из-за операции замыкания он не может даже определить, *сколько* символов нужно просмотреть до обнаружения несовпадения. Чтобы преодолеть такие проблемы, мы наделим автомат свойством *недетерминированности*: на развилке, где возможно несколько способов сопоставления с образцом, машина может "угадать" правильный! Вы можете подумать, что такое свойство невозможно реализовать, но скоро мы увидим, что несложно написать программу для построения *недетерминированного конечного автомата* (НКА) и эффективно моделировать его работу. На высоком уровне алгоритм сопоставления с РВ почти совпадает с алгоритмом Кнута-Морриса-Пратта.

- Создается НКА, соответствующий заданному РВ.
- Моделируется работа этого НКА в заданном тексте.

Теорема Клина (Kleene) — фундаментальный результат теоретических компьютерных наук — утверждает, что для любого заданного РВ существует соответствующий НКА (и наоборот). Мы рассмотрим конструктивное доказательство этого факта, который продемонстрирует, как любое РВ можно преобразовать в НКА, а затем моделировать работу этого НКА, чтобы выполнить непосредственную работу.

Прежде чем мы начнем учиться строить НКА для сопоставления с образцом, мы рассмотрим пример, иллюстрирующий их свойства и основные правила их работы.

На рис. 5.4.1 показан НКА, который определяет, принадлежит ли текстовая строка языку, описанному регулярным выражением `((A*B|AC)D)`. Определенный подобным образом НКА обладает перечисленными ниже характеристиками.

- НКА, соответствующий РВ длиной M, содержит точно одно состояние для каждого символа образца, начинает работу в состоянии 0 и содержит (виртуальное) допускающее состояние M.
- Состояние, соответствующее символу из алфавита, содержит исходящее ребро, которое указывает на состояние, соответствующее следующему символу образца (черные ребра на диаграмме).
- Состояние, соответствующее метасимволу `(`, `)`, `|` или `*`, содержит, по крайней мере, одно исходящее ребро, которое может указывать на любое другое состояние (серые ребра на диаграмме).
- Некоторые состояния могут иметь более одного исходящего ребра, но ни одно состояние не содержит более одного черного исходящего ребра.

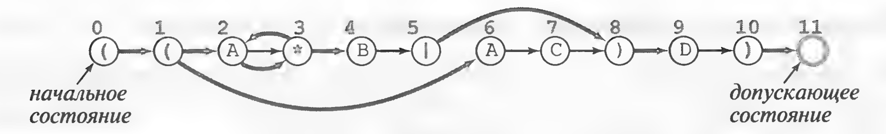


Рис. 5.4.1. НКА, соответствующий образцу `((A*B|AC)D)`

По соглашению мы будем заключать все образцы в скобки, поэтому первое состояние соответствует левой скобке, а конечное состояние соответствует правой скобке (и содержит переход в допускающее состояние).

Как и в случае ДКА из предыдущего раздела, НКА начинает свою работу из состояния 0 и читает первый символ текста. Далее НКА переходит из состояния в состояние, иногда читая символы текста по одному слева направо. Но есть и существенные отличия от ДКА.

- Символы на диаграммах находятся в узлах, а не на ребрах.
- НКА распознает текстовую строку только после явного прочтения всех ее символов, а ДКА распознает образец в тексте, не обязательно прочтя все символы текста.

Эти различия не принципиальны: мы просто выбрали вариант каждой машины, которые лучше подходит для изучаемых нами алгоритмов.

Теперь нужно проверить, что текст соответствует образцу, а для этого нужно, чтобы машина достигла допускающего состояния и прочла весь текст. Правила переходов из одного состояния в другое также отличаются от правил для ДКА. НКА может работать одним из двух способов.

- Если текущее состояние соответствует символу из алфавита *и* текущий символ из текстовой строки соответствует этому символу, автомат может передвинуться на следующий символ в текстовой строке и пройти по (черному) переходу в следующее состояние. Мы будем называть такой переход *переходом соответствия*.
- Автомат может перейти по любому красному ребру в другое состояние без сдвига на следующий символ текста. Мы будем называть такой переход ∈-переходом, поскольку это как бы “соответствие” с пустой строкой ∈.

Предположим, к примеру, что наш НКА для образца `((A*B|AC)D)` запущен (из состояния 0) с входным текстом `AAAABD`. На рис. 5.4.2 показана последовательность переходов между состояниями, которая завершается допускающим состоянием. Эта последовательность демонстрирует, что текст принадлежит множеству строк, описываемых заданным РВ — т.е. текст *соответствует* образцу. В терминах теории автоматов мы говорим, что НКА *распознал* текст.

```
     A     A     A     A     B        D
0→1→2→3→2→3→2→3→2→3→4→5→8→9→10→11
```

переход соответствия: сдвиг на следующий входной символ и изменение состояния

∈-переход: изменение состояния без соответствия

достигнуто допускающее состояние и просмотрены все символы: НКА распознал текст

Рис. 5.4.2. Поиск образца с помощью НКА `((A*B|AC)D)`

Примеры, приведенные на рис. 5.4.3, демонстрируют, что возможны и такие последовательности переходов, которые приводят НКА в тупик, даже для входного текста AAAABD, который он должен распознать. Например, если НКА пройдет по переходу в состояние 4 до просмотра всех A, то ему некуда будет двигаться дальше, т.к. единственным выходом из состояния 4 является сопоставление с B. Эти два примера демонстрируют недетерминированную природу автомата. После просмотра A автомат находится в состоянии 3 и может выбрать один из двух вариантов: перейти в состояние 4 или вернуться в состояние 2. Эти варианты означают разницу между возможностью попасть в допускающее состояние (как в первом примере) и возможностью зависнуть в тупике (как во втором примере). Кроме того, НКА должен сделать выбор в состоянии 1 между ϵ-переходами в состояние 2 или 6.

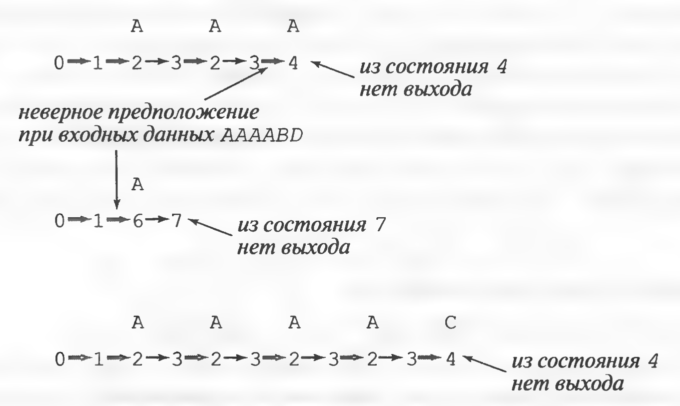


Рис. 5.4.3. *Тупиковые последовательности для НКА* ((A*B|AC)D)

Эти примеры демонстрируют ключевое различие между НКА и ДКА: в НКА из каждого состояния могут выходить несколько ребер, и переход из такого состояния *не определен*: в один момент времени может быть выбран один переход, а в другой момент — другой, без просмотра какого-либо символа текста. Чтобы придать осмысленность действиям такого автомата, представьте себе, что НКА имеет возможность *угадать* переход (если он возможен), который приведет в допускающее состояние для заданной текстовой строки. То есть *НКА распознает текстовую строку тогда и только тогда, когда существует некоторая последовательность переходов, которая просматривает все символы текста и приводит в допускающее состояние, начав с начала текста в состоянии 0.* И наоборот: НКА не распознает текстовую строку тогда и только тогда, когда не существует последовательность переходов и ϵ-переходов, которая может просмотреть все строки этого текста и привести в допускающее состояние.

Как и в случае ДКА, для прослеживания работы НКА с текстовой строкой мы будем просто записывать последовательность изменений состояний, которая заканчивается завершающим состоянием. Любая такая последовательность является доказательством, что машина распознает текстовую строку (могут быть и другие доказательства). Но как найти такую последовательность для заданной текстовой строки? И как доказать, что не существует такой последовательности для другой заданной текстовой строки? Ответы на эти вопросы проще, чем кажется: нужно систематически опробовать все возможности.

Моделирование НКА

Идея автомата, который может угадать переходы между состояниями, необходимые для попадания в допускающее состояние, аналогична программе, которая может угадать правильный ответ задачи — это просто смешно. Но если поразмыслить, то окажется, что концептуально задача не так уж сложна: можно гарантированно проверить все возможные последовательности переходов между состояниями, и если существует такой, который приводит в допускающее состояние, то он будет обнаружен.

Представление

Для начала нам нужно представление НКА. Здесь все просто: состояния нумеруются на основе РВ (целые число от `0` до `M`, где `M` — количество символов в РВ). Само РВ хранится в массиве `re[]` значений `char`, который определяет переходы соответствия (если `re[i]` присутствует в алфавите, то существует переход соответствия из `i` в `i+1`). Естественным представлением для ϵ-переходов является *орграф*: в нем есть направленные ребра, соединяющие вершины с номерами от `0` до `M` (по одному для каждого состояния). Поэтому все ϵ-переходы можно представить в виде орграфа G. Мы рассмотрим задачу построения орграфа, соответствующего заданному РВ, после того как рассмотрим процесс моделирования. Для нашего примера орграф состоит из девяти ребер

```
0→1  1→2  1→6  2→3  3→2  3→4  5→8  8→9  10→11
```

Моделирование НКА и достижимость

Для моделирования работы НКА мы будем отслеживать *множество* состояний, которые могли бы быть достигнуты при просмотре автоматом текущего входного символа. Основное вычисление здесь — уже знакомая нам задача *достижимости из нескольких источников*, которую решает алгоритм 4.4 (листинг 4.2.2). Для инициализации этого множества нужно найти множество состояний, достижимых с помощью ϵ-переходов из состояния 0. Для каждого из этих состояний мы проверяем, возможен ли переход соответствия для первого входного символа. Эта проверка даст множество возможных состояний для НКА сразу после сопоставления первого входного символа. К этому множеству мы добавим все состояния, достижимые с помощью ϵ-переходов из одного из состояний, уже присутствующих в множестве. При наличии множества возможных состояний для НКА после сопоставления первого символа из входных данных решение задачи достижимости из нескольких источников для орграфа ϵ-переходов дает множество состояний, которые могут привести к переходами соответствия для *второго* символа входных данных. Например, начальное множество состояний для нашего демонстрационного НКА равно `0 1 2 3 4 6`; если первый символ равен `A`, НКА выберет переход соответствия в состояние `3` или `7`; затем он может выполнить ϵ-переходы из `3` в `2` или из `3` в `4`, поэтому множество возможных состояний, которое может привести к переходу соответствия для второго символа, равно `2 3 4 7`. Повторение этого процесса до исчерпания всех символов текста приведет к одному из двух возможных завершений:

- множество возможных состояний содержит допускающее состояние;
- множество возможных состояний не содержит допускающее состояние.

Первый их этих вариантов означает, что существует некоторая последовательность переходов, которая переводит НКА в допускающее состояние, и можно сообщить об успешном завершении. А второй вариант означает, что для этих входных данных НКА так

или иначе зависает в тупике, и можно сообщить о неудачном завершении. Наш тип данных SET и класс DirectedDFS, только что описанный для вычисления достижимости в орграфе из нескольких источников, позволяют без труда перевести такое словесное описание в код моделирования НКА, приведенный в листинге 5.4.1. Понимание этого кода можно проверить с помощью трассировки на рис. 5.4.4, где выполнено полное моделирование для нашего примера.

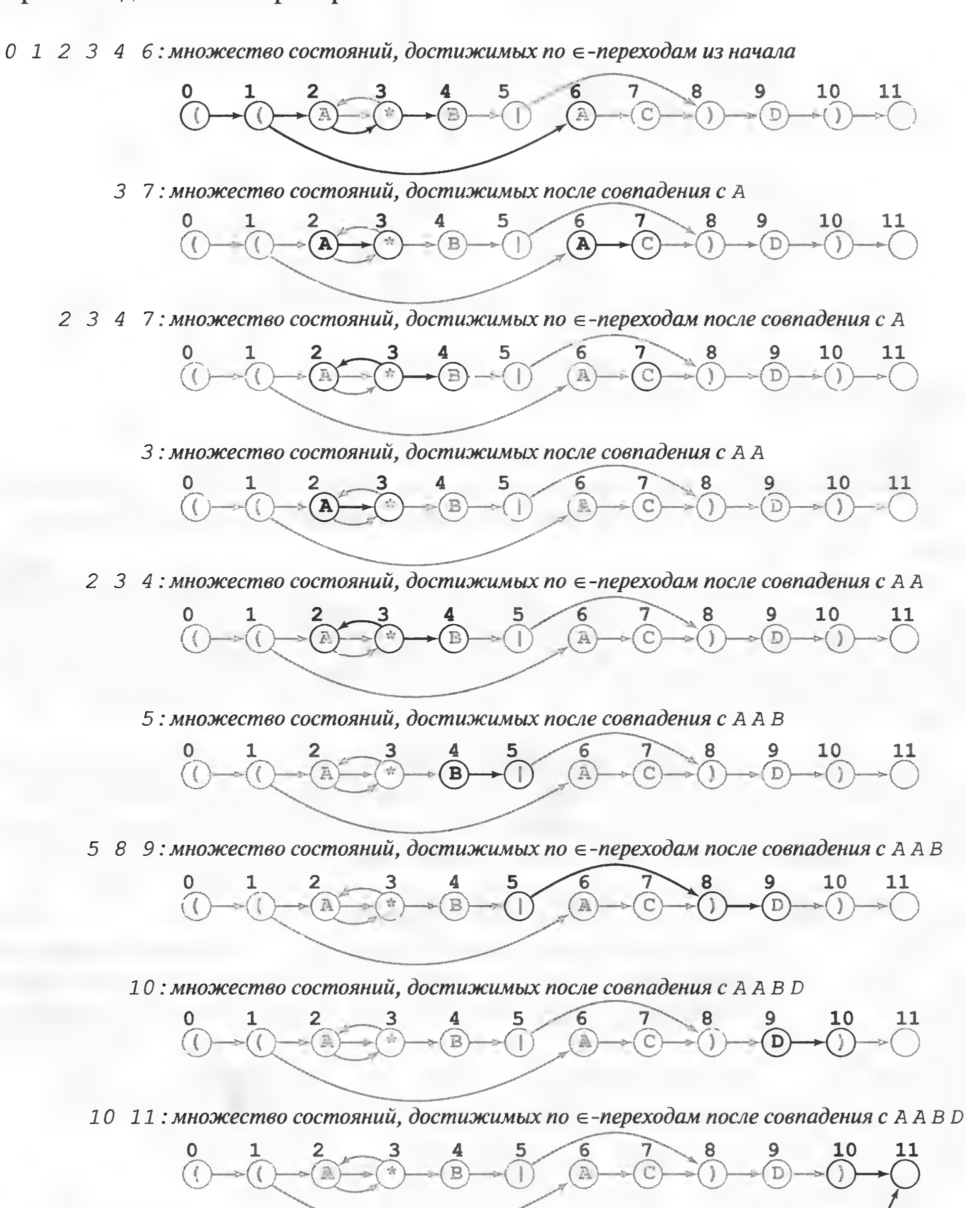


Рис. 5.4.4. *Моделирование работы НКА ((A*B|AC)D) для строки AABD*

Листинг 5.4.1. Моделирование НКА для сравнения с образцом

```
public boolean   recognizes(String txt)
{  // Распознает ли НКА текст txt?
   Bag<Integer> pc = new Bag<Integer>();
   DirectedDFS dfs = new DirectedDFS(G, 0);
   for (int v = 0; v < G.V(); v++)
      if (dfs.marked(v)) pc.add(v);
   for (int i = 0; i < txt.length(); i++)
   {  // Вычисление возможных состояний НКА для txt[i+1].
      Bag<Integer> match = new Bag<Integer>();
      for (int v : pc)
         if (v < M)
            if (re[v] == txt.charAt(i) || re[v] == '.')
               match.add(v+1);
      pc = new Bag<Integer>();
      dfs = new DirectedDFS(G, match);
      for (int v = 0; v < G.V(); v++)
         if (dfs.marked(v)) pc.add(v);
   }
   for (int v : pc) if (v == M) return true;
   return false;
}
```

Утверждение С. Для распознавания N-символьной текстовой строки с помощью НКА, соответствующего M-символьному РВ, в худшем случае требуется время, пропорциональное NM.

Доказательство. Для каждого из N символов текста просматривается множество состояний размером не более M и выполняется поиск в глубину на орграфе ϵ-переходов. Рассмотренное ниже построение устанавливает, что количество ребер в таком орграфе не больше $2M$, поэтому худший случай для каждого поиска в глубину пропорционален M.

Задумайтесь на минутку над этим примечательным результатом. Эта стоимость худшего случая — произведение длин текста и образца — *совпадает* со стоимостью в худшем случае для поиска точного совпадения с подстрокой с помощью элементарного алгоритма, который был рассмотрен в начале раздела 5.3.

Построение НКА, соответствующего РВ

Неудивительно, что в силу сходства регулярных выражений и обычных арифметических выражений трансляция РВ в НКА также похожа на процесс вычисления арифметического выражения с помощью алгоритма Дейкстры с двумя стеками, который был рассмотрен в разделе 1.3.

Однако есть и небольшие отличия:

- в РВ нет явной операции для конкатенации;
- в РВ есть унарная операция для замыкания;
- в РВ только одна бинарная операция — *или* (|).

Но мы не будем разбираться в различиях и сходствах, а просто рассмотрим реализацию, которая специально приспособлена для регулярных выражений. Например, в ней используется не два стека, а только один.

Из обсуждения представления в начале предыдущего подраздела понятно, что нам нужно лишь построить орграф G, который состоит их всех ϵ-переходов. Всю необходимую для этого информацию содержит само РВ и формальные определения, рассмотренные в начале данного раздела. Мы возьмем за основу алгоритм Дейкстры и используем стек, чтобы хранить позиции левых скобок и операций *или*.

Конкатенация

В терминологии НКА операцию конкатенации реализовать проще всего. Переход соответствия для состояний, соответствующих символам алфавита, явно реализуют конкатенацию.

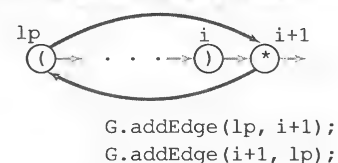
Замыкание одного символа

i i+1

A *

```
G.addEdge(i, i+1);
G.addEdge(i+1, i);
```

Выражение с замыканием



Выражение с операцией или

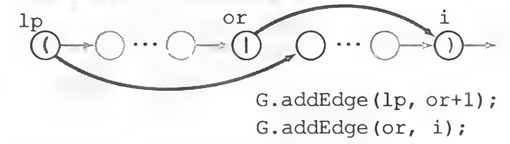


Рис. 5.4.5. *Правила построения НКА*

Скобки

Индекс каждой левой скобки в РВ заносится в стек. Каждый раз, когда попадается правая скобка, из стека извлекается соответствующая левая скобка — так, как описано ниже. Как и в алгоритме Дейкстры, стек позволяет естественно обрабатывать вложенные скобки.

Замыкание

Операция замыкания (*) должна находиться или (1) после отдельного символа, когда добавляются ϵ-переходы в символ и из него, или (2) после правой скобки, когда добавляются ϵ-переходы в соответствующую левую скобку и из нее — в ту, которая находится на верхушке стека.

Выражение ИЛИ

Для обработки РВ вида `(A | B)`, где `A` и `B` — также РВ, к ним добавляются два ϵ-перехода: один из состояния, соответствующего левой скобке, в состояние, соответствующее первому символу `B`, и один из состояния, соответствующего операции |, в состояние, соответствующее правой скобке. Индекс РВ, соответствующий операции `|`, заталкивается в стек (как сказано выше, наряду с индексом, соответствующим левой скобке), и поэтому в момент достижения правой скобки вся необходимая информация находится наверху стека. Эти ϵ-переходы позволяют НКА выбрать один из двух вариантов. Мы не добавляем ϵ-переход из состояния, соответствующего операции `|`, в состояние со следующим более старшим индексом, как для всех других состояний: единственный выход для НКА из такого состояния — переход в состояние, соответствующее правой скобке.

Этих простых правил хватает для построения НКА, соответствующего РВ произвольной сложности. Алгоритм 5.9 (см. листинг 5.4.2) представляет собой реализацию, в которой конструктор строит орграф ϵ-переходов, соответствующий заданному РВ. Трассировка построения для нашего примера приведена на рис. 5.4.7. А на рис. 5.4.5, 5.4.6 и в упражнениях приведены и другие примеры, которые могут повысить ваше понимание процессов на основе проработанных примеров.

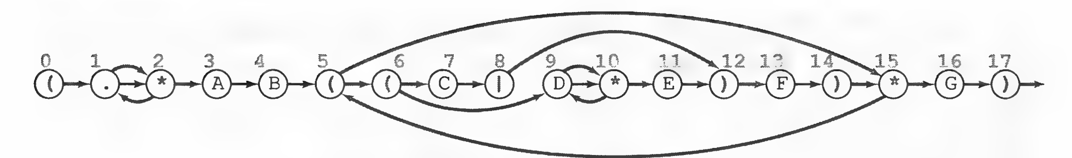


Рис. 5.4.6. НКА, соответствующий образцу (.*AB((C|D*E)F)*G)

стек для индексов левых скобок и операций ИЛИ (ops)

Рис. 5.4.7. Построение НКА, соответствующего образцу ((A*B|AC)D)

Для краткости и ясности некоторые детали (обработка метасимволов, описатели множества символов, сокращения для замыканий и многочастные операции *или*) вынесены в упражнения (см. упражнения 5.4.16–5.4.21). Но основное построение требует очень немного кода и представляет один из наиболее интеллектуальных алгоритмов, которые нам довелось видеть.

Листинг 5.4.2. Алгоритм 5.9. Сравнение образца с регулярными выражениями (grep)

```
public class NFA
{
   private char[] re;                      // переходы соответствия
   private Digraph G;                      // ϵ-переходы
   private int M;                          // количество состояний

   public NFA(String regexp)
   {  // Создание НКА для заданного регулярного выражения.
      Stack<Integer> ops = new Stack<Integer>();
      re = regexp.toCharArray();
      M = re.length;
      G = new Digraph(M+1);

      for (int i = 0; i < M; i++)
      {
         int lp = i;
         if (re[i] == '(' || re[i] == '|')
            ops.push(i);
         else if (re[i] == ')')
         {
            int or = ops.pop();
            if (re[or] == '|')
            {
               lp = ops.pop();
               G.addEdge(lp, or+1);
               G.addEdge(or, i);
            }
            else lp = or;
         }
         if (i < M-1 && re[i+1] == '*')    // заглядывание вперед
         {
            G.addEdge(lp, i+1);
            G.addEdge(i+1, lp);
         }
         if (re[i] == '(' || re[i] == '*' || re[i] == ')')
            G.addEdge(i, i+1);
      }
   }

   public boolean recognizes(String txt)
   // Распознает ли НКА текст txt? (См. листинг 5.4.1.)
}
```

Этот конструктор строит НКА, соответствующий заданному РВ, создавая орграф ϵ-переходов.

Утверждение Т. Построение НКА, соответствующего M-символьному РВ, требует в худшем случае времени и памяти, пропорциональных M.

Доказательство. Для каждого из M символов из регулярного выражения добавляется не более трех ϵ-переходов и, возможно, выполняется одна или две операции в стеке.

Классический клиент GREP для сопоставления с образцом, приведенный в листинге 5.4.3, принимает РВ в качестве аргумента и выводит в стандартный вывод строки, которые содержат некоторые *подстроки*, принадлежащие языку, который описан регулярным выражением. Этот клиент был мощным и незаменимым инструментом для поколений программистов, начиная с ранних реализаций Unix.

Листинг 5.4.3. Классический клиент для НКА обобщенного сравнения с регулярными выражениями (GREP)

```
public class  GREP
{
   public static void main(String[] args)
   {
      String regexp = "(.*" + args[0] + ".*)";
      NFA nfa = new NFA(regexp);
      while (StdIn.hasNextLine())
      {
         String txt = StdIn.hasNextLine();
         if (nfa.recognizes(txt))
            StdOut.println(txt);
      }
   }
}
```

```
% more tinyL.txt
AC
AD
AAA
ABD
ADD
BCD
ABCCBD
BABAAA
BABBAAA
% java GREP "(A*B|AC)D" < tinyL.txt
ABD
ABCCBD
% java GREP StdIn < GREP.java
   while (StdIn.hasNextLine())
      String txt = StdIn.hasNextLine();
```

Вопросы и ответы

Вопрос. **Чем различаются `null` и ϵ?**

Ответ. Первое обозначение применяется для пустого *множества*, а второе — для пустой *строки*. Множество может содержать один элемент ϵ и поэтому не быть пустым.

Упражнения

5.4.1. Приведите регулярные выражения, описывающие все строки, которые содержат:

а) точно четыре последовательных `A`;

б) не более четырех последовательных `A`;

в) по крайней мере одно вхождение четырех последовательных `A`.

5.4.2. Приведите краткое описание для каждого из следующих РВ:

а) `.*`

б) `A.*A | A`

в) `.*ABBABBA.*`

г) `.* A.*A.*A.*A.*`

5.4.3. Каково максимальное количество различных строк, которые можно описать регулярным выражением с M операциями *или* и без операций замыкания (скобки и конкатенация допускаются)?

5.4.4. Нарисуйте НКА, соответствующий образцу `(((A|B)*|CD*|EFG)*)*`.

5.4.5. Нарисуйте орграф ϵ-переходов для НКА из упражнения 5.4.4.

5.4.6. Приведите множество состояний, достижимых для НКА из упражнения 5.4.4 после совпадения с каждым символом, и последующие ϵ-переходы для входных данных `ABBACEFGEFGCAAB`.

5.4.7. Напишите на основе клиента `GREP` из листинга 5.4.3 клиент `GREPmatch`, который заключает образец в скобки, но *не* добавляет `.*` перед образцом и после него — тогда он будет выводить только строки, которые входят в язык, описанный заданным РВ. Приведите результат выполнения следующих команд:

а) `% java GREPmatch "(A|B)(C|D)" < tinyL.txt`

б) `% java GREPmatch "A(B|C)*D" < tinyL.txt`

в) `% java GREPmatch "(A*B|AC)D" < tinyL.txt`

5.4.8. Напишите регулярные выражения для каждого из следующих множеств двоичных строк:

а) содержит не менее трех последовательных единиц;

б) содержит подстроку `110`;

в) содержит подстроку `1101100`;

г) не содержит подстроку `110`.

5.4.9. Напишите регулярное выражение для двоичных строк, которые содержат не менее двух нулей, но не содержат последовательных нулей.

5.4.10. Напишите регулярное выражение для каждого из следующих множеств двоичных строк:

а) содержат не менее трех символов, причем третий символ — `0`;

б) количество нулей кратно 3;

в) начинаются и оканчиваются одним и тем же символом;

г) нечетной длины;

д) начинаются с `0` и имеют нечетную длину или начинаются с `1` и имеют четную длину;

е) длина не меньше 1 и не больше 3.

5.4.11. Для каждого из следующих регулярных выражений укажите, сколько двоичных строк длиной точно 1000 символов соответствуют им:

а) `0(0 | 1)*1`

б) `0*101*`

в) `(1 | 01)*`

5.4.12. Напишите регулярные выражения для каждого из следующего множества строк:

а) телефонные номера наподобие `(609) 555-1234`;

б) номера карточек социального страхования наподобие `123-45-6789`;

в) даты вроде `December 31, 2012`;

г) IP-адреса в виде `a.b.c.d`, где каждая буква может представлять одну, две или три цифры, наподобие `196.26.155.241`;

д) автомобильные номера, которые начинаются четырьмя цифрами и оканчиваются двумя прописными буквами.

Творческие задачи

5.4.13. *Сложные РВ.* Придумайте РВ, которые описывают каждое из следующих множеств строк из двоичного алфавита:

а) все строки, кроме `11` и `111`;

б) строки с единицами в каждой нечетной битовой позиции;

в) строки с не менее двумя нулями и не более чем одной единицей;

г) строки без двух единиц подряд.

5.4.14. *Двоичная делимость.* Придумайте РВ, описывающие все двоичные строки, которые, если рассматривать их как двоичные числа, делятся на:

а) 2

б) 3

в) 123

5.4.15. *Одноуровневые РВ.* Напишите РВ, описывающее множество строк, которые допустимы в двоичном алфавите, но без вложенных скобок. Например, строки (0.*1)* и (1.*0)* принадлежат этому языку, а (1(0 и 1)1)* — нет.

5.4.16. *Многочастное или.* Добавьте в НКА многочастные операции *или*. Ваш код должен сгенерировать для образца (.*AB((C|D|E)F)*G) машину, приведенную на рис. 5.4.8.

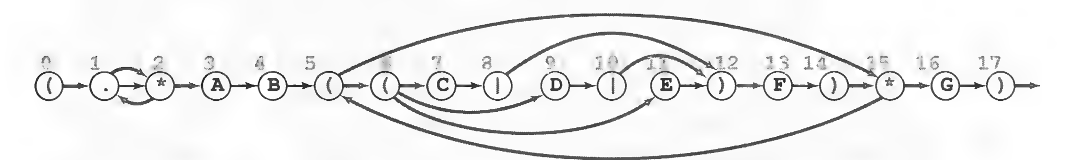


Рис. 5.4.8. *НКА, соответствующий образцу* (.*AB((C|D|E)F)*G)

5.4.17. *Обобщенные символы.* Добавьте в класс NFA обработку обобщенных символов.

5.4.18. *Один или несколько.* Добавьте в класс NFA обработку операции замыкания +.

5.4.19. *Заданное множество.* Добавьте в класс NFA обработку описателей заданного множества.

5.4.20. *Диапазон.* Добавьте в класс NFA обработку описателей диапазонов.

5.4.21. *Дополнение.* Добавьте в класс NFA обработку описателей дополнения.

5.4.22. *Доказательство.* Напишите версию НКА, выводящую *доказательство*, что заданная строка принадлежит языку, который распознается этим НКА (последовательность переходов между состояниями, завершающаяся допускающим состоянием).

5.5. Сжатие данных

Мир буквально захлебывается данными, и алгоритмы представления данных играют важную роль в современной вычислительной инфраструктуре. Для сжатия данных есть две основных причины — экономия памяти при хранении информации и экономия времени при обмене информацией. Обе эти причины остаются важными уже много технологических поколений и знакомы сегодня каждому, кто подыскивает новое устройство для хранения данных или ожидает окончания длительной загрузки.

Вы наверняка встречались со сжатием при работе с цифровыми изображениями, звуками, фильмами и всякими другими данными. Алгоритмы, с которыми мы ознакомимся, экономят память за счет того, что большинство файлов данных содержат значительную избыточность данных. Например, в текстовых файлах некоторые последовательности символов встречаются значительно чаще других; растровые изображения содержат большие области одного цвета; а файлы, представляющие в цифровом виде изображения, фильмы, звуки и другие аналоговые сигналы содержат длинные повторяющиеся фрагменты.

Мы рассмотрим элементарный алгоритм и два более сложных метода, нашедших широкое практическое применение. Сжатие, которое достигается этими методами, обычно дает 20–50% экономии, а в некоторых ситуациях и 50–90%. Как мы увидим, эффективность любого метода сжатия данных весьма зависит от характеристик входных данных. *Примечание*: когда в этой книге речь идет о производительности, мы обычно говорим о времени, но при обсуждении сжатия данных мы, как правило, говорим о степени полученного сжатия — хотя, конечно, нас интересует и время, необходимое для выполнения работы.

С одной стороны, технологии сжатия данных сейчас не так актуальны, как раньше, поскольку стоимость компьютерных запоминающих устройств существенно снизилась, и в распоряжении типичного пользователя находится значительно больший объем памяти, чем раньше. А с другой стороны, технологии сжатия данных сейчас актуальны как никогда раньше, т.к. при использовании огромных массивов информации экономия может быть гораздо значительнее, чем раньше. Вообще говоря, сжатие данных получило наибольшее распространение с возникновением Интернета, поскольку оно представляет собой дешевый способ снизить время, необходимое для передачи больших объемов данных.

Сжатие данных имеет богатую историю (мы приведем лишь краткое введение в эту тему), и, несомненно, стоит подумать над его ролью в будущем. Каждому изучающему теорию алгоритмов будет полезно разобраться со сжатием данных, ведь оно выполняется классическими, элегантными, интересными и эффективными алгоритмами.

Правила игры

Все типы данных, которые обрабатываются в современных компьютерных системах, имеют нечто общее: *в конечном счете они представлены в двоичном виде*. И каждый из них можно считать просто последовательностью битов (или байтов). Для краткости в данном разделе мы будем использовать термин *битовый поток*, понимая под ним биты, организованные в последовательность байтов фиксированного размера. Битовый или байтовый поток можно хранить на компьютере в виде файла или передавать по сети в виде сообщения.

Базовая модель

Поэтому наша базовая модель для сжатия данных довольно проста (рис. 5.5.1) и содержит два основных компонента, которые читают и записывают битовые потоки.

- Компонент *упаковки*, который преобразовывает битовый поток B в сжатую версию $C(B)$.
- Компонент *распаковки*, который преобразовывает $C(B)$ обратно в B.

Если обозначить количество битов в битовом потоке как $|B|$, то мы заинтересованы в минимизации величины $|C(B)| / |B|$, которая называется *коэффициентом сжатия* (compression ratio).

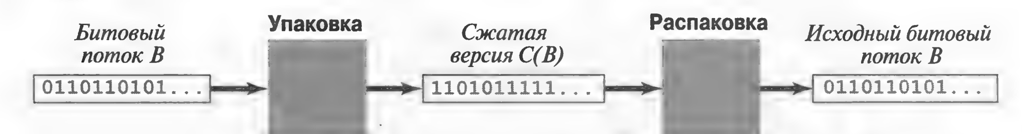


Рис. 5.5.1. Базовая модель для сжатия данных

Эта модель называется *сжатием без потерь*: мы требуем, чтобы информация не терялась, в том смысле, что результат упаковки и распаковки битового потока должен совпадать с исходным потоком вплоть до последнего бита. Сжатие без потерь требуется для многих типов данных, таких как числовые данные или исполняемый код. Для некоторых типов данных (таких как изображения, видео или музыка) можно рассмотреть методы сжатия, которые позволяют терять некоторую часть информации, и тогда декодер формирует лишь приблизительный вариант исходного файла. Методы сжатия с потерей информации приходится оценивать, кроме коэффициента сжатия, еще и в соответствии с неким субъективным стандартом качества. В данной книге мы не будем рассматривать такое сжатие.

Чтение и запись двоичных данных

Полное описание кодирования информации в вашем компьютере зависит от системы и выходит за рамки данной книги. Однако несколько естественных предположений и два простых API позволят отделить наши реализации от этих деталей. Эти API — `BinaryStdIn` и `BinaryStdOut` — похожи на привычные нам API `StdIn` и `StdOut`, но они предназначены для чтения *битов*, в то время как `StdIn` и `StdOut` ориентированы на *потоки символов* в кодировке Unicode. Значение типа `int` в `StdOut` представляет собой последовательность символов (его десятичное представление), а значение типа `int` в `BinaryStdOut` представляет собой последовательность битов (его двоичное представление).

Бинарный ввод и вывод

Большинство современных систем, включая и Java, работают с вводом-выводом в виде потоков 8-битовых байтов, поэтому естественно читать и записывать байтовые потоки в соответствии с форматом внутреннего представления примитивных типов: кодировать 8-битовый тип `char` одним байтом, 16-битовый тип `short` — двумя байтами, 32-битовый `int` — четырьмя и т.д. Поскольку *битовые потоки* — это основная абстракция для сжатия данных, мы пойдем немного дальше и разрешим клиентам читать и записывать отдельные *биты* вперемешку с данными примитивных типов. Мы хотим минимизировать объем преобразования типов в клиентских программах и в то же время не на-

рушать системные соглашения по представлению данных. Для чтения битового потока из стандартного ввода мы будем использовать API, представленный на рис. 5.5.2.

public class **BinaryStdIn**		
boolean	readBoolean()	*чтение 1 бита данных и возврат в виде значения* boolean
char	readChar()	*чтение 8 битов данных и возврат в виде значения* char
char	readChar(int r)	*чтение* r *(от 1 до 16) битов данных и возврат в виде значения* char
[*аналогичные методы для* byte *(8 битов);* short *(16 битов);* int *(32 бита);* long *и* double *(64 бита)*]		
boolean	isEmpty()	*пуст ли битовый поток?*
void	close()	*закрытие битового потока*

Рис. 5.5.2. *API для статических методов чтения из битового потока через стандартный ввод*

Основное отличие этой абстракции от StdIn в том, что *данные в стандартном вводе не обязательно выровнены по границе байта*. Если входной поток содержит один байт, клиент может прочитать его по одному биту, восемь раз вызвав метод readBoolean(). Метод close() не так важен, но для аккуратного завершения клиенты должны вызывать его, чтобы указать, что чтение битов прекращается. Как и в случае пары StdIn/StdOut, мы вводим аналогичный API для записи битовых потоков в стандартный вывод, показанный на рис. 5.5.3.

public class **BinaryStdOut**		
void	write(boolean b)	*запись указанного бита*
void	write(char c)	*запись указанного 8-битового* char
void	write(char c, int r)	*запись* r *(от 1 до 16) младших битов указанного* char
[*аналогичные методы для* byte *(8 битов);* short *(16 битов);* int *(32 бита);* long *и* double *(64 бита)*]		
void	close()	*закрытие битового потока*

Рис. 5.5.3. *API для статических методов записи в битовый поток через стандартный вывод*

При выводе данных метод close() *обязателен*: клиенты должны вызывать его, чтобы все биты, указанные в предыдущих вызовах write(), попали в битовый поток, а завершающий байт был дополнен нулями для выравнивания по границе байта в соответствии с требованиями файловой системы. Аналогично API In и Out, связанным с StdIn и StdOut, мы используем API BinaryIn и BinaryOut, которые позволяют обращаться непосредственно к файлам в двоичной кодировке.

Пример

В качестве простого примера предположим, что имеется тип данных, где данные представляются тремя значениями int — месяц, день и год. Запись таких значений в формате 12/31/2012 потребует 10 символов, т.е. 80 битов. Если записать эти значения

непосредственно в BinaryStdOut, это займет даже 96 битов (по 32 бита для каждого из трех значений int). Если использовать более экономичное представление — значения byte для номера месяца и дня и short для года — то хватит 32 битов. А с помощью BinaryStdOut можно записать 4-битовое поле, 5-битовое поле и 12-битовое поле — всего 21 бит (реально 24 бита, т.к. общая длина должна содержать целое количество 8-битовых байтов, и метод close() дописывает в конец три нулевых бита) (рис. 5.5.4). *Важное примечание*: такая экономия уже грубо иллюстрирует принцип сжатия данных.

Двоичные дампы

Как можно просмотреть содержимое битового или байтового потока на этапе отладки? Этот вопрос был актуален уже давно, когда единственным способом отладки был просмотр каждого бита памяти, и термин *дамп* использовался с ранних дней компьютерных вычислений для описания представления битового потока, доступного для просмотра человеком. Если попробовать открыть файл в редакторе или просмотреть его как текстовый файл (или даже просто запустить программу, которая работает с BinaryStdOut), вы наверняка увидите какие-то бессмысленные символы, разные в различных системах. Класс BinaryStdIn позволяет избежать подобной зависимости от системы: для этого нужно написать собственные программы для преобразования битовых потоков в наглядное представление. К примеру, программа BinaryDump, приведенная в листинге 5.5.1, представляет собой клиент класса BinaryStdIn, выводящий биты из стандартного ввода в виде символов 0 и 1. Эта программа удобна для отладки при работе с небольшими объемами входных данных. Аналогичный клиент HexDump группирует данные в 8-битовые байты и выводит каждый байт в виде двух шестнадцатеричных цифр, каждая из которых представляет 4 бита. Клиент PictureDump выводит биты в виде изображения Picture, где нулевые биты представляют белые пиксели, а единичные биты — черные пиксели. Такое графическое представление часто полезно для выявления повторяющихся мест в битовых потоках (рис. 5.5.5).

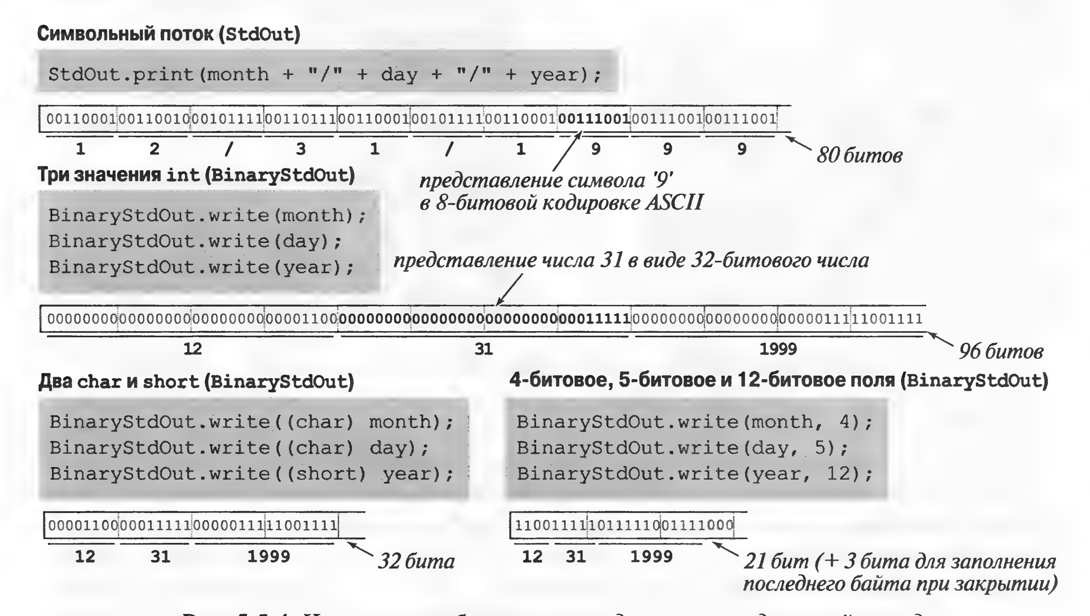


Рис. 5.5.4. *Четыре способа помещения даты в стандартный вывод*

Программы `BinaryDump`, `HexDump` и `PictureDump` можно загрузить с сайта книги. При работе с двоичными файлами обычно мы используем конвейер и перенаправление на уровне командной строки: выходные данные кодирующей программы передаются программе `BinaryDump`, `HexDump` или `PictureDump`, либо перенаправляются в файл.

Листинг 5.5.1. Вывод битового потока в стандартный (символьный) вывод

```
public class BinaryDump
{
   public static void main(String[] args)
   {
      int width = Integer.parseInt(args[0]);
      int cnt;
      for (cnt = 0; !BinaryStdIn.isEmpty(); cnt++)
      {
         if (width == 0) continue;
         if (cnt != 0 && cnt % width == 0)
            StdOut.println();
         if (BinaryStdIn.readBoolean())
              StdOut.print("1");
         else StdOut.print("0");
      }
      StdOut.println();
      StdOut.println(cnt + " bits");
   }
}
```

Стандартный символьный поток

```
% more abra.txt
ABRACADABRA!
```

Представление битового потока символами 0 и 1

```
% java BinaryDump 16 < abra.txt
0100000101000010
0101001001000001
0100001101000001
0100010001000001
0100001001010010
0100000100100001
96 bits
```

Представление битового потока шестнадцатеричными цифрами

```
% java HexDump 4 < abra.txt
41 42 52 41
43 41 44 41
42 52 41 21
96 bits
```

Представление битового потока в виде пикселей объекта `Picture`

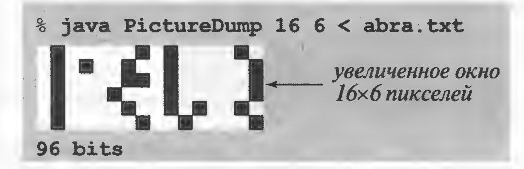


Рис. 5.5.5. Четыре способа просмотра битового потока

Кодировка символами ASCII

При работе с шестнадцатеричными дампами, содержащими символы в кодировке ASCII, для справочных целей может пригодится таблица, приведенная в табл. 5.5.1. Если символ закодирован двумя шестнадцатеричными цифрами, используйте для определения закодированного символа первую шестнадцатеричную цифру в качестве номера строки, а вторую — в качестве номера столбца. Например, число `31` кодирует цифру `1`, число `4A` — букву `J` и т.д. Эта таблица составлена для 7-битовых ASCII-символов, поэтому

первая цифра не должна превышать 7. Шестнадцатеричные числа, начинающиеся с 0 и 1 (а также числа 20 и 7F), соответствуют непечатаемым управляющим символам. Многие управляющие символы появились в те времена, когда физические устройства вроде принтеров управлялись входными ASCII-символами, и в таблице представлены те из них, которые можно увидеть в дампе. Например, SP — пробел, NUL — нулевой символ, LF — символ перевода строки, а CR — символ возврата каретки.

Таблица 5.5.1. Преобразование шестнадцатеричных чисел в символы ASCII

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	`	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

В общем, работа со сжатием данных требует переориентации мышления со стандартного ввода и вывода на бинарное кодирование. API BinaryStdIn и BinaryStdOut содержат все необходимые для этого методы. Они позволяют четко разделить клиентские программы для вывода информации, предназначенной для хранения в файле или передачи другим программам, и для вывода информации, предназначенной для чтения людьми.

Ограничения

Чтобы разбираться в алгоритмах сжатия данных, необходимо знать фундаментальные ограничения. Для этого исследователи разработали исчерпывающий и важный теоретический фундамент, который мы кратко рассмотрим в конце данного раздела, но для начала достаточно несколько принципов.

Универсальное сжатие данных

Вооружившись алгоритмическими средствами, которые доказали свою пользу при решении столь многих задач, вы можете подумать, что наша цель — *универсальное сжатие данных*, т.е. алгоритм, уменьшающий объем произвольного битового потока. А вот и нет: мы поставим более скромные цели, т.к. универсальное сжатие данных невозможно.

Утверждение У. Ни один алгоритм не может сжать произвольный битовый поток.

Доказательство. Мы рассмотрим два доказательства, каждое из которых позволит лучше понять предмет нашего изучения. Первое из них — от противного: предположим, что имеется такой алгоритм, который сжимает любой битовый поток. Тогда этот алгоритм можно использовать для сжатия его выходных данных и получить еще более короткий битовый поток, и продолжать так, пока длина потока не станет нулевой (рис. 5.5.6)! Вывод, что алгоритм может сжать любой битовый поток до 0 битов, абсурден, и поэтому абсурдно предположение о возможности сжатия произвольного битового потока.

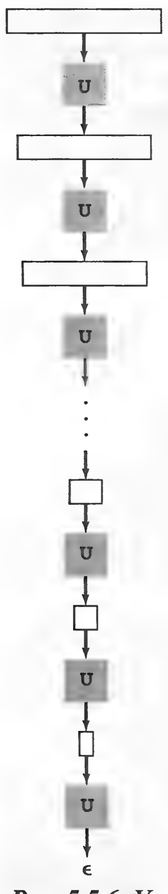


Рис. 5.5.6. Универсальное сжатие данных?

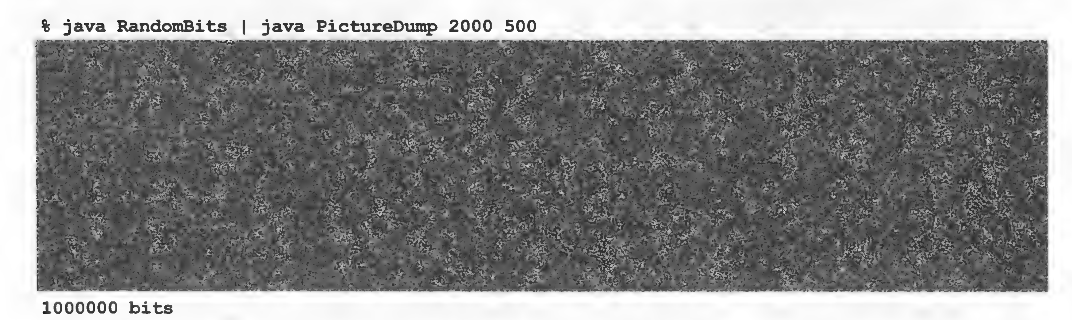
Второе доказательство основано на подсчете. Пусть имеется алгоритм, который выполняет сжатие без потери информации для любого 1000-битового потока. То есть каждый такой поток должен отображаться на другой более короткий поток. Однако существует лишь $1 + 2 + 4 + \ldots + 2^{998} + 2^{999} = 2^{1000} - 1$ битовых потоков объемом меньше 1000 битов и 2^{1000} битовых потоков из 1000 битов, поэтому алгоритм не может сжать каждый из них. Это рассуждение выглядит более убедительным, если рассматривать более жесткие требования. Допустим, требуется получить коэффициент сжатия лучше 50%. Имейте в виду, что это может получиться лишь для одного из примерно 2^{500} битовых потоков длиной 1000 битов!

Иначе говоря, имеется не более одного из 2^{500} шансов сжать наполовину случайный 1000-битовый поток с помощью любого алгоритма сжатия данных. Если вы узнаете о каком-то новом алгоритме сжатия без потери информации, наверняка он не сможет заметно сжать случайный битовый поток. Понимание, что не стоит рассчитывать на сжатие случайных битовых потоков, лежит в основе понимания сжатия данных. Регулярно выполняется обработка строк из миллионов или миллиардов битов, но никогда не будет обработана даже малая часть всех возможных таких строк, поэтому не надо огорчаться этим теоретическим результатом. Ведь битовые строки, которые мы обычно обрабатываем, сильно структурированы, и этот факт используется для сжатия данных.

Неразрешимость

Посмотрите на строку из миллиона битов на рис. 5.5.7. Эта строка с виду совершенно случайная, поэтому вряд ли вы подберете алгоритм ее сжатия без потери информации. Однако ее можно представить всего лишь несколькими тысячами битов, потому что она порождена программой, приведенной в листинге 5.5.2. (Это пример генератора псевдослучайных чисел наподобие Java-метода `Math.random()`.) Алгоритм сжатия, который сможет сгенерировать программу в ASCII-коде, а затем выполнить ее, получит коэффициент сжатия 0,3%, который трудно превзойти — а этот коэффициент можно уменьшить еще более, подав на вход больше битов!

```
% java RandomBits | java PictureDump 2000 500
```



```
1000000 bits
```

Рис. 5.5.7. Файл, трудный для сжатия: 1 миллион (псевдо)случайных битов

Для сжатия такого файла необходимо найти программу, которая генерирует его. Этот пример далеко не так отвлечен, как кажется на первый взгляд: при сжатии видео, или отсканированной старой книги, или какого-то другого файла, которых полно в Интернете, мы всегда что-то узнаем и о программе, с помощью которой был получен этот файл.

Понимание, что бо́льшая часть обрабатываемых данных порождена какими-то программами, приводит к фундаментальным вопросам в теории вычислений — а также дает некоторое представление о трудностях при сжатии данных. Например, можно доказать, что оптимальное сжатие данных (нахождение самой короткой программы, порождающей заданную строку) является *неразрешимой* задачей: у нас не только нет алгоритма, который сжимает произвольный битовый поток, но нет даже и стратегии для разработки наилучшего алгоритма!

Листинг 5.5.2. "Сжатый" поток из миллиона битов

```
public class RandomBits
{
   public static void main(String[] args)
   {
      int x = 11111;
      for (int i = 0; i < 1000000; i++)
      {
         x = x * 314159 + 218281;
         BinaryStdOut.write(x > 0);
      }
      BinaryStdOut.close();
   }
}
```

Практическое следствие этих ограничений таково: методы сжатия без потери информации должны быть ориентированы на использование *известной* структуры сжимаемых битовых потоков. Четыре метода, которые мы рассмотрим, используют следующие структурные характеристики:

- небольшие алфавиты;
- длинные последовательности одинаковых битов или символов;
- часто используемые символы;
- длинные повторяющиеся последовательности битов или символов.

Если известно, что заданный битовый поток обладает одной или несколькими из этих характеристик, можно сжать его одним из методов, с которыми вы познакомитесь ниже; а если нет, то, возможно, все же стоит попытаться использовать их, т.к. базовая структура данных вполне может быть неочевидной, а эти методы пригодны во многих случаях. Вы увидите, что у каждого метода есть параметры и разновидности, которые позволяют выполнить настройку на лучшее сжатие конкретного битового потока. Первое и единственное, что нужно сделать — выяснить структуру данных и использовать это знание для сжатия, возможно, с помощью одной из технологий, которые мы сейчас рассмотрим.

Разминка: геномика

В качестве подготовки к изучению более сложных алгоритмов сжатия данных мы сейчас рассмотрим элементарную (но очень важную) задачу. Во всех наших реализациях будут использоваться одни и те же соглашения, которые мы поясним в контексте данного примера.

Геномные данные

В первом примере сжатия данных рассмотрим следующую строку:

ATAGATGCATAGCGCATAGCTAGATGTGCTAGCAT

Использование стандартной кодировки ASCII (1 байт или 8 битов на символ) преобразовывает эту строку в битовый поток длиной $8 \times 35 = 280$ битов. Подобные строки очень важны в современной биологии, поскольку биологи используют буквы `A`, `C`, `T` и `G` для представления четырех нуклеотидов в составе ДНК живых организмов. *Геном* — это последовательность нуклеотидов. Ученые знают, что выяснение свойств геномов представляет собой ключ к пониманию процессов, происходящих в живых организмах — включая жизнь, смерть и болезни. Геномы для многих живых существ уже известны, и ученые пишут программы для изучения структуры таких последовательностей.

Сжатие 2-битовым кодом

У геномов есть замечательное свойство: они состоят лишь из четырех различных символов, поэтому каждый из них можно закодировать двумя битами, как в методе `compress()` из листинга 5.5.3. Хотя входные данные кодируются текстовыми символами, для их чтения мы используем класс `BinaryStdIn`, чтобы подчеркнуть приверженность к стандартной модели сжатия данных (битовый поток в битовый поток). Количество закодированных символов также помещается в сжатый файл, чтобы правильно выполнить распаковку, если последний бит не попадает на границу байта. Программа преобразовывает каждый 8-битовый символ в 2-битовый код и добавляет еще 32 бита для хранения длины, поэтому она достигает коэффициента сжатия 25% по отношению к длинным входным последовательностям символов.

Листинг 5.5.3. Метод упаковки для геномных данных

```
public static void compress()
{
   Alphabet DNA = new Alphabet("ACTG");
   String s = BinaryStdIn.readString();

   int N = s.length();
   BinaryStdOut.write(N);

   for (int i = 0; i < N; i++)
   {  // Записать двухбитовый код для символа.
      int d = DNA.toIndex(s.charAt(i));
      BinaryStdOut.write(d, DNA.lgR());
   }

   BinaryStdOut.close();
}
```

Распаковка 2-битового кода

Метод expand() из листинга 5.5.4 выполняет распаковку битового потока, полученного методом compress(). Как и в случае упаковки, этот метод читает битовый поток и записывает битовый поток, в соответствии с базовой моделью сжатия данных. При этом в качестве выходных данных получаются первоначальные данные.

Листинг 5.5.4. Метод распаковки для геномных данных

```
public static void expand()
{
   Alphabet DNA = new Alphabet("ACTG");
   int w = DNA.lgR();
   int N = BinaryStdIn.readInt();
   for (int i = 0; i < N; i++)
   {  // Прочитать 2 бита; записать символ.
      char c = BinaryStdIn.readChar(w);
      BinaryStdOut.write(DNA.toChar(c));
   }
   BinaryStdOut.close();
}
```

Этот подход работает и для других алфавитов фиксированного размера, но мы оставим это обобщение в качестве (легкого) упражнения (см. упражнение 5.5.25).

Рассмотренные методы не вполне соответствуют стандартной модели сжатия данных, т.к. сжатый битовый поток не содержит всю информацию, необходимую для его распаковки. Тот факт, что алфавит состоит только из букв A, C, T и G, является соглашением между двумя методами. Подобное соглашение разумно в приложениях наподобие геномики, где один и тот же код может использоваться постоянно. В других ситуациях алфавит может потребоваться включить в упакованное сообщение (см. упражнение 5.5.25). Такие затраты также учитываются при сравнении методов сжатия данных.

На заре развития геномики изучение геномных последовательностей было долгим и трудоемким занятием, поэтому последовательности были относительно короткими, и для их записи и обмена ученые использовали стандартную ASCII-кодировку. С тех пор экспериментальный процесс был значительно упрощен, и теперь известно много геномов огромной длины (геном человека содержит более 10^{10} битов информации), поэтому 75% экономии, которую дают эти простые методы — весьма значительный объем. Есть ли возможность для дальнейшего сжатия? Это очень интересный вопрос, потому что это *научный* вопрос: возможность сжатия означает наличие некой структуры в данных, а основной целью современной геномики является обнаружение структуры в геномных данных. Стандартные методы сжатия данных вроде тех, которые мы рассмотрим ниже, неэффективны для геномных данных (в 2-битовой кодировке), как для случайных данных.

Методы compress() и expand() мы оформляем в виде статических методов одного класса и добавим туда же простой драйвер, приведенный в листинге 5.5.5. Чтобы проверить свое понимание правил игры и базовых инструментов, применяемых для сжатия данных, обязательно разберитесь в различных командах, приведенных на рис. 5.5.8, которые выполняют вызовы Genome.compress() и Genome.expand() для наших демонстрационных данных (и в результатах этих вызовов).

Маленький тестовый пример (264 бита)

```
% more genomeTiny.txt
ATAGATGCATAGCGCATAGCTAGATGTGCTAGC

java BinaryDump 64 < genomeTiny.txt
0100000101010100010000010100011101000001010101000100011101000011
0100000101010100010000010100011101000011010001110100001101000001
0101010001000001010001110100001101010100010000010100011101000001
0101010001000111010101000100011101000011010101000100000101000111
01000011
264 bits

% java Genome - < genomeTiny.txt
?? ←—— битовый поток не виден
       на стандартном выводе

% java Genome - < genomeTiny.txt | java BinaryDump 64
0000000000000000000000000010000100100011001011010010001101110100
1000110110001100101110110110001101000000
104 bits

% java Genome - < genomeTiny.txt | java HexDump 8
00 00 00 21 23 2d 23 74
8d 8c bb 63 40
104 bits

% java Genome - < genomeTiny.txt > genomeTiny.2bit
% java Genome + < genomeTiny.2bit
ATAGATGCATAGCGCATAGCTAGATGTGCTAGC

% java Genome - < genomeTiny.txt | java Genome +
ATAGATGCATAGCGCATAGCTAGATGTGCTAGC
```

цикл упаковки-распаковки дает исходные данные

Реальный вирус (50 000 битов)

```
% java PictureDump 512 100 < genomeVirus.txt
```

50000 bits

```
% java Genome - < genomeVirus.txt | java PictureDump 512 25
```

12536 bits

Рис. 5.5.8. *Упаковка и распаковка геномных последовательностей с помощью 2-битового кодирования*

Листинг 5.5.5. Соглашение по упаковке для методов сжатия данных

```
public class Genome
{
   public static void compress()
   // См. текст.

   public static void expand()
   // См. текст.

   public static void main(String[] args)
   {
      if (args[0].equals("-")) compress();
      if (args[0].equals("+")) expand();
   }
}
```

Кодирование по длинам серий

Простейшим видом избыточности в битовом потоке являются длинные последовательности одинаковых битов. Сейчас мы рассмотрим классический метод, называемый *кодированием по длинам серий* (run-length encoding), который использует эту избыточность для сжатия данных. Например, рассмотрим следующую строку из 40 битов:

```
0000000000000001111111000000011111111111
```

Эта строка состоит из 15 нулей, 7 единиц, 7 нулей и 11 единиц, поэтому ее можно закодировать числами 15, 7, 7 и 11. Все битовые строки состоят из чередующихся серий нулей и единиц, и достаточно просто указывать длины таких серий. Если кодировать числа четырьмя битами и начать с серии нулей, то получится 16-битовая строка

```
1111011101111011
```

(15 = `1111`, затем 7 = `0111`, затем 7 = `0111`, и затем 11 = `1011`), что дает коэффициент сжатия 16/40 = 40%. Чтобы превратить это описание в эффективный метод сжатия данных, необходимо рассмотреть следующие вопросы.

- Сколько битов использовать для хранения счетчиков?
- Что делать, если встретится серия, длина которой больше максимального значения счетчика, обусловленного предыдущим выбором?
- Что делать с сериями, которые короче, чем количество битов, необходимое для хранения их длины?

Нас в основном интересуют длинные битовые потоки с относительно короткими сериями, поэтому мы выберем следующие ответы на эти вопросы.

- Счетчики могут иметь значения от 0 до 255 и кодируются 8 битами.
- Все длины серий мы делаем короче 256, для чего при необходимости вставляются серии нулевой длины.
- Короткие серии тоже кодируются, даже если это увеличит длину выходных данных.

Этот выбор легко реализовать, и он весьма эффективен для нескольких видов битовых потоков, широко распространенных на практике. Он *не* эффективен при наличии множества коротких серий: биты экономятся лишь на сериях, длины которых больше количества битов, необходимых для их хранения.

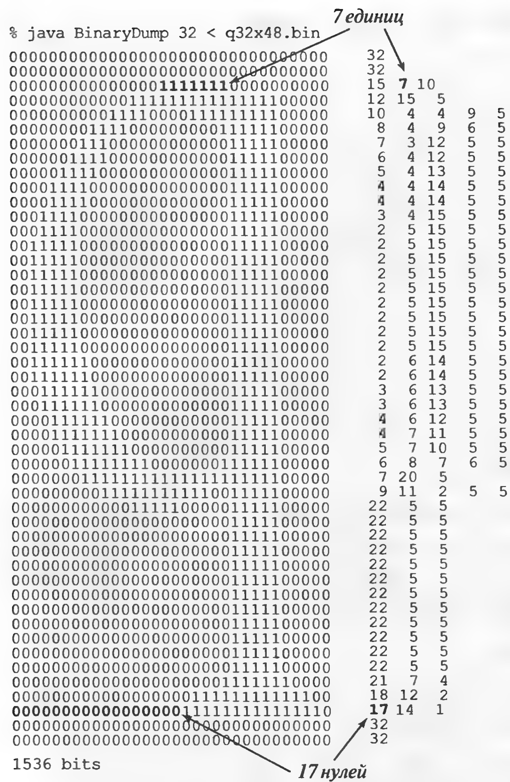


Рис. 5.5.9. *Типичное битовое изображение с длинами серий для каждой строки*

Растровые изображения

В качестве примера эффективности кодирования по длинам серий мы рассмотрим *растровые изображения* (bitmap), которые широко используются для хранения картинок и отсканированных документов. Для краткости и простоты мы рассмотрим растровые изображения с битовыми значениями, организованные в виде битовых потоков из строк пикселей. Для просмотра таких изображений мы используем программу `PictureDump`. Несложно написать программу для преобразования изображения из одного из многих распространенных форматов хранения изображений без потери информации, которые применяются для экранных снимков или отсканированных документов, в битовое отображение. Наш пример для демонстрации эффективности кодирования по длинам серий использует экранные снимки данной книги — а именно, изображение буквы *q* в различных разрешениях. В основном мы будем пользоваться двоичным дампом изображения размером 32×48 пикселей, который приведен на рис. 5.5.9 вместе с длинами серий для каждой строки. Поскольку каждая строка начинается и заканчивается нулями, в каждой строке содержится нечетное количество длин серий; поскольку за одной строкой сразу же начинается следующая, соответствующая длина серии нулевых битов в битовом потоке равна сумме длины последней серии в предыдущей строке и длины первой серии в следующей строке (с некоторыми добавлениями, которые соответствуют строкам, состоящим только из нулей).

Реализация

Из приведенного неформального описания непосредственно следуют реализации методов `compress()` и `expand()`, приведенные в листинге 5.5.6. Как обычно, реализация `expand()` проще: она читает длину серии, выводит столько копий текущего бита, изменяет значение бита на противоположное и продолжает до окончания входных данных. Метод `compress()` ненамного сложнее; он выполняет следующие шаги, пока во входном потоке имеются биты.

- Чтение бита.
- Если он отличается от предыдущего бита — запись текущего счетчика и затем обнуление счетчика.
- Если он совпадает с предыдущим битом и счетчик содержит максимальное значение — запись счетчика, запись нулевого счетчика и обнуление счетчика.
- Увеличение счетчика на единицу.

После прочтения всего входного потока процесс завершается записью текущего счетчика (длины последней серии).

Листинг 5.5.6. Методы распаковки и упаковки для кодирования по длинам серий

```
public static void  expand()
{
   boolean b = false;
   while (!BinaryStdIn.isEmpty())
   {
      char cnt = BinaryStdIn.readChar();
      for (int i = 0; i < cnt; i++)
         BinaryStdOut.write(b);
      b = !b;
   }
   BinaryStdOut.close();
}

public static void compress()
{
   char cnt = 0;
   boolean b, old = false;
   while (!BinaryStdIn.isEmpty())
   {
      b = BinaryStdIn.readBoolean();
      if (b != old)
      {
         BinaryStdOut.write(cnt);
         cnt = 0;
         old = !old;
      }
      else
      {
         if (cnt == 255)
         {
            BinaryStdOut.write(cnt);
            cnt = 0;
            BinaryStdOut.write(cnt);
         }
      }
      cnt++;
   }
   BinaryStdOut.write(cnt);
   BinaryStdOut.close();
}
```

Увеличение разрешения в растровых изображениях

Основная причина широкого распространения кодирования по длинам серий для растровых изображений состоит в значительном увеличении эффективности при увеличении разрешения. В этом нетрудно убедиться. Пусть для нашего примера разрешение увеличено вдвое. Тогда очевидно, что:

- количество битов увеличивается в 4 раза;
- количество серий увеличивается примерно в два раза;
- длины серий увеличиваются примерно в два раза;
- количество битов в сжатой версии увеличивается примерно в два раза;
- значит, коэффициент сжатия уменьшается в два раза!

Без кодирования при удвоении разрешения объем необходимой памяти увеличивается в 4 раза, а с кодированием по длинам строк объем памяти увеличивается только в 2 раза. То есть с увеличением разрешения объем памяти и коэффициент сжатия линейно уменьшаются. Например, для нашей буквы q с низким разрешением получается коэффициент сжатия лишь 74%, а при увеличении разрешения до 64×96 коэффициент падает до 37%. Это изменение наглядно видно на примере выходных данных программы `PictureDump`, которые приведены на рис. 5.5.10. Буква в более высоком разрешении занимает в четыре раза больше памяти по сравнению с низким разрешением (удвоение по двум измерениям), однако сжатая версия занимает лишь вдвое больше памяти (удвоение по одному измерению). Если еще увеличить разрешение до 128×192 (примерно столько нужно для печати), то коэффициент падает до 18% (см. упражнение 5.5.5).

Сжатие Хаффмана

Теперь мы рассмотрим технологию сжатия данных, которая может дать значительную экономию памяти в файлах на естественных языках (а также во многих других видах файлов). Она основана на изменении принципа хранения текстовых файлов: вместо использования обычных 7 или 8 битов для каждого символа используется меньше битов для символов, которые встречаются чаще других.

Чтобы пояснить этот принцип, рассмотрим небольшой пример. Пусть необходимо закодировать строку `ABRACADABRA!`. 7-битовая кодировка ASCII дает такую битовую строку:

```
100000110000101010010100000110000111000001-
100010010000011000010101001010000010100001.
```

Для декодирования этой битовой строки мы просто считываем поочередно 7 битов и преобразовываем их в соответствии с таблицей ASCII, приведенной в табл. 5.5.1. В этой стандартной кодировке буква `D`, которая встречается только один раз, требует для хранения столько же битов, что и буква `A`, которая встречается пять раз. Сжатие Хаффмана основано на экономии битов, которая достигается кодированием часто используемых символов меньшим количеством битов по сравнению с редко встречающимися символами, что уменьшает общее количество требуемых битов.

Беспрефиксные коды переменной длины

Код связывает с каждым символом некоторую битовую строку — т.е. это таблица имен с символами в качестве ключей и битовыми строками в качестве значений. Для начала можно попробовать назначить самые короткие битовые строки наиболее распространенным символам: букве `A` — строку `0`, `B` — `1`, `R` — `00`, `C` — `01`, `D` — `10` и символу `!` — `11`. Тогда строка `ABRACADABRA!` будет закодирована в виде `0 1 00 0 01 0 10 0 1 00 0 11`. Такое представление содержит только 17 битов (по сравнению с 7-битовой кодировкой ASCII), но это все-таки не код, потому что для него необходимы пробелы, разделяющие закодированные символы.

Маленький тестовый пример (40 битов)

```
% java BinaryDump 40 < 4runs.bin
0000000000000001111111000000011111111111
40 bits

% java RunLength - < 4runs.bin | java HexDump
0f 07 07 0b
32 bits        коэффициент сжатия 32/40 = 80%

% java RunLength - < 4runs.bin | java RunLength + | java BinaryDump 40
0000000000000001111111000000011111111111 ←— упаковка-распаковка
40 bits                                       дает исходные данные
```

ASCII-текст (96 битов)

```
% java RunLength - < abra.txt | java HexDump 24
01 01 05 01 01 01 04 01 02 01 01 01 02 01 02 01 05 01 01 01 04 02 01 01
05 01 01 01 03 01 03 01 05 01 01 01 04 01 02 01 01 01 02 01 02 01 05 01
02 01 04 01   коэффициент сжатия 416/96 = 433% —
416 bits ←—   не используйте кодирование по длинам серий для ASCII!
```

Растровое изображение (1536 битов)

```
% java RunLength - < q32x48.bin > q32x48.bin.rle
% java HexDump 16 < q32x48.bin.rle
4f 07 16 0f 0f 04 04 09 0d 04 09 06 0c 03 0c 05
0b 04 0c 05 0a 04 0d 05 09 04 0e 05 09 04 0e 05
08 04 0f 05 08 04 0f 05 07 05 0f 05 07 05 0f 05
07 05 0f 05 07 05 0f 05 07 05 0f 05 07 05 0f 05
07 05 0f 05 07 05 0f 05 07 06 0e 05 07 06 0e 05
08 06 0d 05 08 06 0d 05 09 06 0c 05 09 07 0b 05
0a 07 0a 05 0b 08 07 06 0c 14 0e 0b 02 05 11 05
05 05 1b 05 1b 05 1b 05 1b 05 1b 05 1b 05 1b 05
1b 05 1b 05 1b 05 1b 05 1a 07 16 0c 13 0e 41
1144 bits ←— коэффициент сжатия 1144/1536 = 74%
```

```
% java PictureDump 32 48 <
  q32x48.bin
```

1536 bits

```
% java PictureDump 32 36 <
  q32x48.rle.bin
```



1144 bits

Растровое изображение с большим разрешением (6144 бита)

```
% java BinaryDump 0 < q64x96.bin
6144 bits
% java RunLength - < q64x96.bin | java BinaryDump 0
2296 bits ←— коэффициент сжатия 2296/6144 = 37%
```

```
% java PictureDump 64 96 <
  q64x96.bin
```



6144 bits

```
% java PictureDump 64 36 <
  q64x96.rle.bin
```



2296 bits

Рис. 5.5.10. Упаковка и распаковка битовых потоков с помощью кодирования по длинам серий

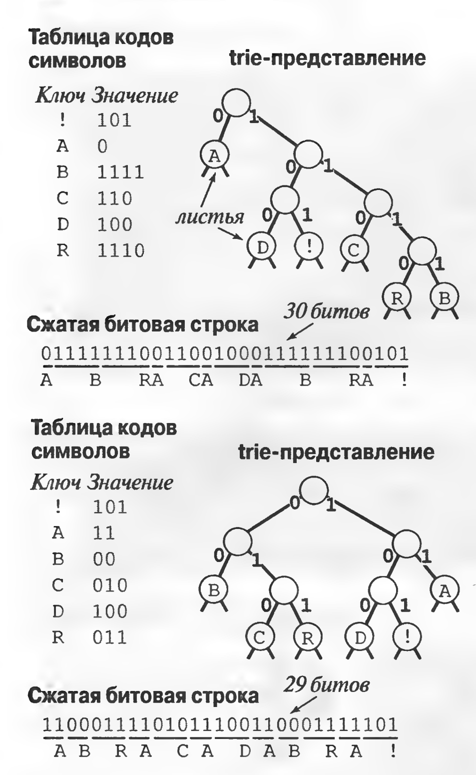


Рис. 5.5.11. Два беспрефиксных кода

Без пробелов битовая строка имеет вид

`01000010100100011`

и ее можно декодировать многими способами — например, `CRRDDCRCB`. Но даже 17 битов плюс 10 разделителей компактнее стандартной кодировки, в основном потому, что не используются биты для кодировки букв, которых нет в сообщении. Однако можно использовать тот факт, что *разделители не нужны, если ни один код символа не является префиксом другого*. Код с таким свойством называется *беспрефиксным* (prefix-free) *кодом*. Приведенная выше кодировка не является беспрефиксной, т.к. код буквы `A` (`0`) является префиксом кода буквы `R` (`00`). Но если, к примеру, закодировать букву `A` битом `0`, `B` — битами `11111`, `C` — `110`, `D` — `100`, `R` — `1110` и символ `!` — битами 101, то 30-битовую строку

`011111110011001000111111100101`

можно декодировать единственным способом — `ABRACADABRA!`. Все беспрефиксные коды *уникально декодируемы* (без каких-либо разделителей) и поэтому широко используются. Все коды фиксированной длины наподобие 7-битового ASCII являются беспрефиксными.

Представление беспрефиксных кодов с помощью trie-деревьев

Одним из удобных способов представления беспрефиксного кода является trie-дерево (см. раздел 5.2). Любое trie-дерево с M нулевыми ссылками определяет беспрефиксный код для M символов: нужно заменить нулевые ссылки ссылками на *листья* (узлы с двумя нулевыми ссылками), каждый из которых содержит кодируемый символ, и определить код для каждого символа как битовую строку, описываемую путем от корня до этого символа. Для trie-деревьев принято бит 0 связывать с перемещением влево, а бит 1 — с перемещением вправо. Например, на рис. 5.5.11 показаны два беспрефиксных кода для символов из строки `ABRACADABRA!`. В верхней части приведен только что рассмотренный код, а в нижней — код, который дает строку

`11000111101011100110001111101`

длиной 29 битов, т.е. на 1 бит короче.

Существует ли trie-дерево, которое дает еще большее сжатие? Как найти trie-дерево, которое определяет наилучший беспрефиксный код? Оказывается, на эти вопросы имеется элегантный ответ в виде алгоритма, вычисляющего trie-дерево, которое приводит к генерации битового потока минимальной длины для любой заданной строки. Для честного сравнения с другими кодами нужно считать и биты самого кода, поскольку без него строку невозможно декодировать, и, как вы увидите, этот код зависит от строки. Общий метод нахождения оптимального беспрефиксного кода был открыт Д. Хаффманом (D. Huffman) в 1952 г. (еще студентом!), и поэтому он называется *кодированием Хаффмана*.

Обзор

Использование беспрефиксного кода для сжатия данных выполняется в пять этапов. Кодируемый битовый поток рассматривается как байтовый поток, и для символов этого потока используется беспрефиксный код.

- Строится кодирующее trie-дерево.
- Это trie-дерево записывается (в виде битового потока) для последующей распаковки.
- С помощью trie-дерева выполняется кодировка байтового потока в битовый поток.

Для распаковки нужно выполнить следующие действия.

- Чтение trie-дерева (закодированного в начале битового потока).
- Декодирование битового потока с помощью этого trie-дерева.

Чтобы лучше разобраться в этом процессе, мы рассмотрим вышеописанные шаги в порядке возрастания сложности.

Узлы trie-дерева

Начнем с определения класса `Node`, показанного в листинге 5.5.7. Он похож на вложенные классы, которые мы использовали раньше для построения бинарных деревьев и trie-деревьев: каждый узел `Node` содержит ссылки `left` и `right` на другие узлы, которые и определяют структуру trie-дерева. В каждом узле также имеется переменная экземпляров `freq`, необходимая для построения дерева, и переменная экземпляров `ch`, которая используется в листьях для представления кодируемых символов.

Листинг 5.5.7. Представление узла trie-дерева

```
private static class  Node implements Comparable<Node>
{  // Удел дерева для кодирования Хаффмана
   private char ch;     // не используется для внутренних узлов
   private int freq;    // не используется на этапе распаковки
   private final Node left, right;

   Node(char ch, int freq, Node left, Node right)
   {
      this.ch    = ch;
      this.freq  = freq;
      this.left  = left;
      this.right = right;
   }

   public boolean isLeaf()
   {  return left == null && right == null;  }

   public int compareTo(Node that)
   {  return this.freq - that.freq;  }
}
```

Распаковка для беспрефиксных кодов

При наличии trie-дерева, определяющего код, распаковка закодированного битового потока выполняется просто. Метод `expand()` из листинга 5.5.8 представляет собой реализацию этого процесса. После чтения trie-дерева из стандартного ввода с помощью метода `readTrie()`, который будет описан ниже, оно используется для распаковки

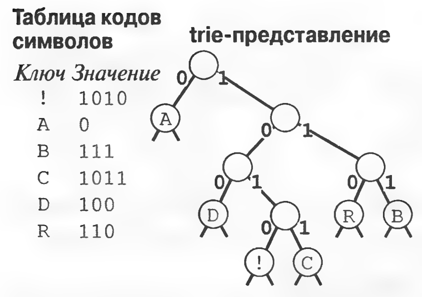


Рис. 5.5.12. Пример кода Хаффмана

остальной части битового потока. Мы начинаем с корня и спускаемся по дереву в соответствии с битами потока: читается очередной входной бит и выполняется переход влево, если он равен 0, или вправо, если он равен 1. При достижении листа выводится символ из этого узла, и процесс продолжается опять с корня дерева. Если вы ознакомитесь с действием этого метода на небольшом примере кода, представленном на рис. 5.5.12, вы поймете и уясните этот процесс. Например, для декодирования битовой строки 011111001011... мы начинаем с корня, переходим влево, т.к. первый бит равен 0, выводим символ A; возвращаемся в корень, три раза переходим вправо, выводим символ B; возвращаемся в корень, два раза переходим вправо, затем влево, затем выводим R; и т.д. Простота распаковки является одной из причин популярности беспрефиксных кодов вообще и сжатия Хаффмана в частности.

Листинг 5.5.8. Распаковка беспрефиксного кода (декодирование)

```
public static void expand()
{
   Node root = readTrie();
   int N = BinaryStdIn.readInt();
   for (int i = 0; i < N; i++)
   {  // Распаковка i-го символа.
      Node x = root;
      while (!x.isLeaf())
         if (BinaryStdIn.readBoolean())
              x = x.right;
         else x = x.left;
      BinaryStdOut.write(x.ch);
   }
   BinaryStdOut.close();
}
```

Упаковка для беспрефиксных кодов

Для упаковки используется trie-дерево, определяющее код — см. метод buildCode() в листинге 5.5.9. Это элегантный и компактный метод, но несколько хитроумный, поэтому в нем следует тщательно разобраться. Для любого trie-дерева он генерирует таблицу, которая связывает с каждым символом в дереве битовую строку (в виде переменной String, содержащей символы 0 и 1). Таблица кодирования представляет собой таблицу имен, которая связывает такое значение String с каждым символом. Для эффективности мы используем не таблицу имен общего вида, а индексированный символами массив st[], поскольку количество символов невелико. Для ее создания метод buildCode() рекурсивно проходит по дереву и использует двоичную строку, соответствующую пути от корня до каждого узла (0 для левых ссылок и 1 для правых), а также кодовое слово, соответствующее каждому символу при обнаружении этого символа в листе. После построения таблицы кодирования упаковка не представляет труда: нужно просто выбирать код для каждого символа входных данных. Для кодирования строки ABRACADABRA! с

помощью кода с рис. 5.5.12 мы записываем 0 (строка, связанная с символом A), потом 111 (строка, связанная с символом B), потом 110 (строка, связанная с символом R) и т.д. Эту задачу выполняет фрагмент кода, приведенный в листинге 5.5.10: ищется значение String, связанное с каждым входным символом, преобразовывается в значения 0/1 в массиве типа char, и соответствующая битовая строка записывается в выходной поток.

Листинг 5.5.9. Построение таблицы кодирования из trie-дерева (беспрефиксного) кода

```
private static String[] buildCode(Node root)
{  // Создание таблицы поиска из trie-дерева.
   String[] st = new String[R];
   buildCode(st, root, "");
   return st;
}

private static void buildCode(String[] st, Node x, String s)
{  // Создание таблицы поиска из trie-дерева (рекурсивное).
   if (x.isLeaf())
   {  st[x.ch] = s; return; }
   buildCode(st, x.left,  s + '0');
   buildCode(st, x.right, s + '1');
}
```

Листинг 5.5.10. Сжатие с помощью таблицы кодирования

```
for (int i = 0; i < input.length; i++)
{
   String code = st[input[i]];
   for (int j = 0; j < code.length(); j++)
      if (code.charAt(j) == '1')
           BinaryStdOut.write(true);
      else BinaryStdOut.write(false);
}
```

Построение trie-дерева

Описанный ниже процесс иллюстрируется на рис. 5.5.13 на примере построения дерева Хаффмана для входной строки

```
it was the best of times it was the worst of times
```

Кодируемые символы хранятся в листьях, и в каждом узле переменная экземпляров freq содержит частоту появления всех символов в поддереве с корнем в данном узле. Сначала создается лес деревьев из 1 узла (листьев), по одному для каждого символа из входного потока, и для каждого из них подсчитывается значение freq — счетчик его появления во входном потоке. В нашем примере входные данные содержат 8 букв t, 5 букв e, 11 пробелов и т.д. (*Важное замечание*: для получения этих счетчиков необходимо прочитать весь входной поток — кодирование Хаффмана представляет собой *двухпроходный* алгоритм, т.к. для выполнения сжатия необходимо прочитать входной поток еще раз.) Затем формируется trie-дерево — снизу вверх, на основе счетчиков символов. Во время построения дерева оно рассматривается как бинарное trie-дерево со счетчиками в узлах, а после построения оно используется как дерево для кодирования, как было описано выше.

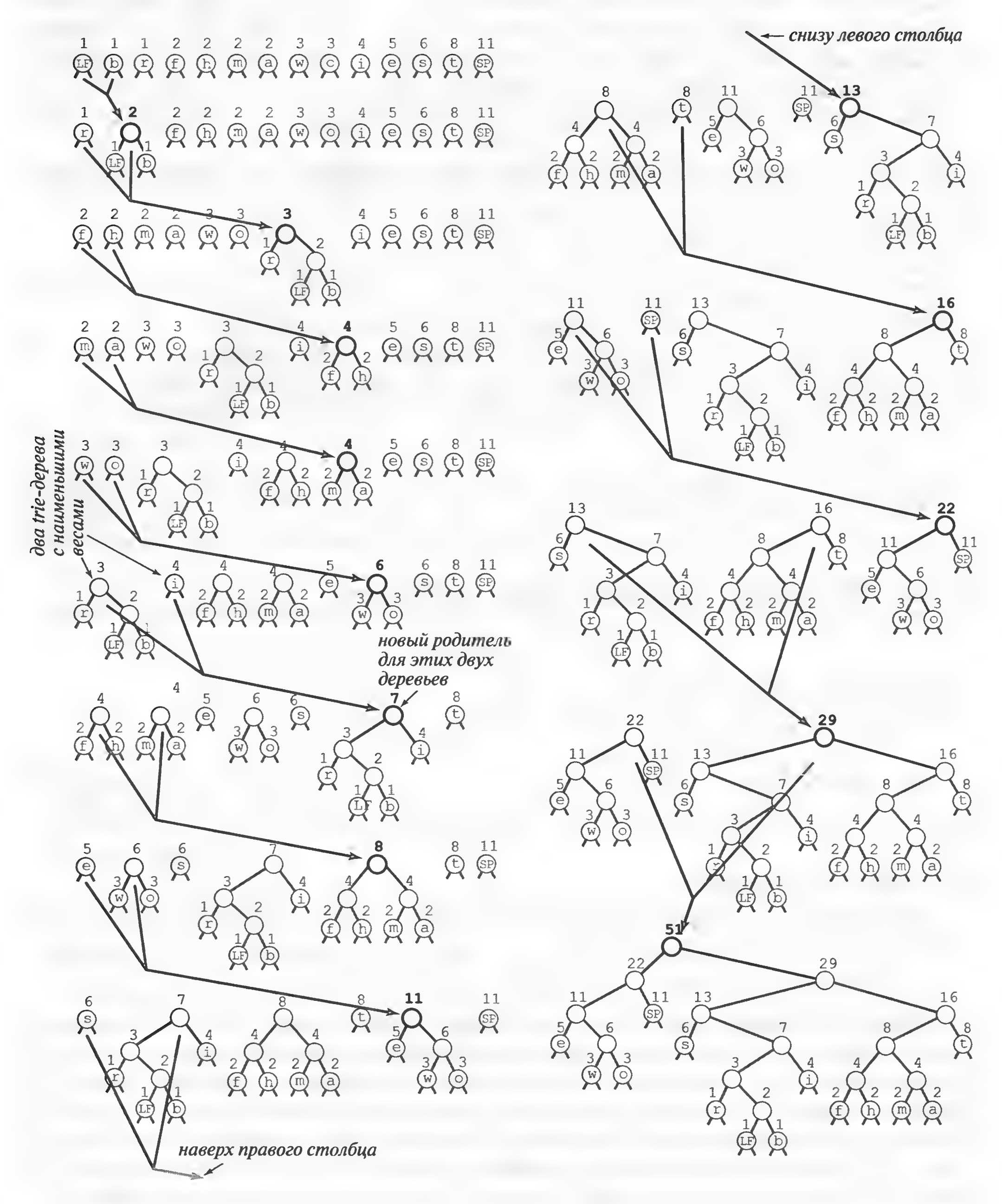


Рис. 5.5.13. Построение trie-дерева кодирования Хаффмана

Процесс работает следующим образом: находятся два узла с наименьшими счетчиками, и создается новый узел, для которого эти узлы являются дочерними (а значение счетчика равно сумме значений счетчиков дочерних узлов). Эта операция уменьшает количество trie-деревьев в лесе на одно. Затем этот процесс повторяется: в лесе находятся два узла с наименьшими счетчиками, и точно так же создается новый узел. Реализация такого процесса естественно выполняется с помощью очереди с приоритетами, как показано в методе `buildTrie()` из листинга 5.5.11. (Для ясности trie-деревья на рис. 5.5.13 всегда упорядочены.) Постепенно создаются все бо́льшие и бо́льшие деревья, и одновременно на каждом шаге на единицу уменьшается их количество (удаляются два, а добавляется одно). В конце концов, все узлы будут объединены в одно дерево. Листья в этом дереве содержат кодируемые символы и частоты их повторений во входных данных, а каждый узел, не являющийся листом, содержит сумму частот его двух дочерних узлов. Узлы с низкими частотами находятся где-то внизу дерева, а узлы с высокими частотами находятся ближе к корню. Значение счетчика в корне равно количеству всех символов во входных данных. Это бинарное trie-дерево, содержащее символы только в листьях, поэтому оно определяет беспрефиксный код для символов. Использование таблицы кодовых последовательностей, созданных методом `buildCode()` для нашего примера (справа на рис. 5.5.14), дает выходную битовую строку

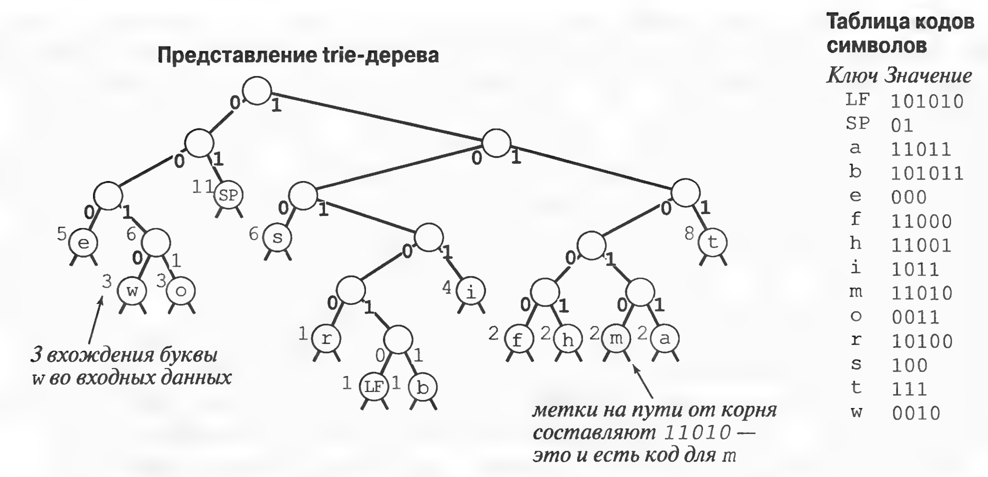
```
10111110100101101110001111110010000110101100-
01001110100111100001111101111010000100011011-
11101001011011100011111100100001001000111010-
01001110100111100001111101111010000100101010.
```

длиной 176 битов — т.е. экономия составляет 57% по сравнению с 408 битами, необходимыми для кодирования 51 символа в стандартной 8-битовой кодировке ASCII (не считая стоимости включения кода, которую мы рассмотрим чуть ниже). Более того, поскольку это код *Хаффмана*, никакой другой беспрефиксный код не может закодировать входные данные меньшим количеством битов (см. утверждение X ниже).

Листинг 5.5.11. Построение trie-дерева кодирования Хаффмана

```
private static Node buildTrie(int[] freq)
{
   // Первоначальное занесение одиночных деревьев в очередь с приоритетами.
   MinPQ<Node> pq = new MinPQ<Node>();
   for (char c = 0; c < R; c++)
      if (freq[c] > 0)
         pq.insert(new Node(c, freq[c], null, null));

   while (pq.size() > 1)
   {  // Слияние двух наименьших деревьев.
      Node x = pq.delMin();
      Node y = pq.delMin();
      Node parent = new Node('\0', x.freq + y.freq, x, y);
      pq.insert(parent);
   }
   return pq.delMin();
}
```



Ключ	Значение
LF	101010
SP	01
a	11011
b	101011
e	000
f	11000
h	11001
i	1011
m	11010
o	0011
r	10100
s	100
t	111
w	0010

Рис. 5.5.14. *Код Хаффмана для символьного потока*
"it was the best of times it was the worst of timesLF"

Оптимальность

Мы уже заметили, что часто повторяющиеся символы находятся ближе к корню дерева, чем редко встречающиеся символы, и поэтому они кодируются меньшим количеством битов. Значит, это хороший код, но почему это *оптимальный* беспрефиксный код? Чтобы ответить на данный вопрос, мы начнем с определения *взвешенной длины внешнего пути* дерева, которая равна сумме весов (равных счетчикам повторений), умноженных на глубину (см. раздел 1.5) всех листьев.

Утверждение Ф. Для любого беспрефиксного кода длина закодированной битовой строки равна взвешенной длине внешнего пути для соответствующего trie-дерева.

Доказательство. Глубина каждого листа равна количеству битов, необходимых для кодирования символа в этом листе. Поэтому взвешенная длина внешнего пути равна длине закодированной битовой строки: она равна сумме по всем символам количества их повторений, умноженных на количество битов для каждого повторения.

В нашем примере имеется один лист на расстоянии 2 (SP с частотой 11), три листа на расстоянии 3 (e, s и t, с общей частотой 19), три листа на расстоянии 4 (w, o и i, с общей частотой 10), пять листьев на расстоянии 5 (r, f, h, m и a, с общей частотой 9) и два листа на расстоянии 6 (LF и b, с общей частотой 2). Общая сумма равна $2 \cdot 11 + 3 \cdot 19 + 4 \cdot 10 + 5 \cdot 9 + 6 \cdot 2 = 176$, и, как и ожидалось, она равна длине выходной битовой строки.

Утверждение Х. Для любого множества *r* символов и частот алгоритм Хаффмана строит оптимальный беспрефиксный код.

Доказательство. По индукции по *r*. Допустим, что код Хаффмана оптимален для любого множества, состоящего из менее *r* символов. Пусть T_H — код, вычисленный алгоритмом Хаффмана для множества символов и связанных с ними частот (s_1, r_1), ..., (s_r, f_r); обозначим длину кода (взвешенную длину внешнего пути) как $W(T_H)$. Предположим, что (s_i, f_i) и (s_j, f_j) — два первых выбранных символа. Тогда алгоритм вычисляет код T_H^* для множества $n-1$ символов, в котором (s_i, f_i) и (s_j, f_j) заменены на $(s^*, f_i + f_j)$, где s^* — новый символ в листе на некоторой глубине d.

Имеет место равенство

$$W(T_H) = W(T_H^*) - d\,(f_i + f_j) + (d + 1)\,(f_i + f_j) = W(T_H^*) + (f_i + f_j)$$

Теперь рассмотрим оптимальное trie-дерево T для множества $(s_1, r_1), ..., (s_r, f_r)$ высотой h. Символы (s_i, f_i) и (s_j, f_j) должны находиться на глубине h, иначе мы могли бы построить trie-дерево с меньшей длиной внешнего пути, обменяв их с узлами на глубине h. Кроме того, предположим, что (s_i, f_i) и (s_j, f_j) являются родственными узлами — для этого достаточно обменять (s_j, f_j) с узлом, родственным (s_i, f_i). Рассмотрим дерево T^*, полученное заменой их родительского узла на $(s^*, f_i + f_j)$. При этом (согласно рассуждению, аналогичному вышеприведенному) $W(T) = W(T^*) + (f_i + f_j)$.

Согласно гипотезе индукции дерево T_H^* оптимально: $W(T_H^*) \le W(T^*)$.

Следовательно,

$$W(T_H) = W(T_H^*) + (f_i + f_j) \le W(T^*) + (f_i + f_j) = W(T)$$

В силу оптимальности T должно выполняться равенство, и код T_H оптимален.

При выборе очередного узла может оказаться, что несколько узлов имеют одинаковый вес. Метод Хаффмана не определяет порядок разбивки таких trie-деревьев. Он также не определяет, какой из дочерних узлов должен находиться слева, а какой справа. Различные выборы приводят к различным кодам Хаффмана, но все такие коды кодируют сообщение в оптимальное количество битов из всех беспрефиксных кодов.

Запись и чтение trie-дерева

Как уже было подчеркнуто, приведенные выше значения экономии не совсем точны, т.к. сжатый битовый поток невозможно декодировать без trie-дерева. Поэтому необходимо учитывать стоимость включения trie-дерева в упакованные выходные данные вместе с самой битовой строкой. Для длинных входных данных эта стоимость относительно мала, но полная схема сжатия данных подразумевает запись trie-дерева в битовый поток при упаковке и его чтение при распаковке. Как можно закодировать trie-дерево в виде битового потока, а потом распаковать его? Обе эти задачи можно выполнить с помощью простых рекурсивных процедур на основе *прямого обхода* дерева.

Процедура `writeTrie()`, приведенная в листинге 5.5.12, выполняет обход trie-дерева в прямом порядке: при посещении внутреннего узла записывается только нулевой бит; при посещении листа записывается единичный бит, а за ним 8-битовый ASCII-код символа в этом листе. Битовая строка, кодирующая дерево Хаффмана для нашего примера `ABRACADABRA!`, приведена на рис. 5.5.15. Первый нулевой бит соответствует корню. Следующим узлом является лист, содержащий `A`, поэтому следующий бит равен `1`, а за ним следует 8-битовый ASCII-код `00100001` для буквы `A`. Следующие два бита равны `0`, т.к. при обходе встретились два внутренних узла, и т.д.

Листинг 5.5.12. Запись trie-дерева в виде битовой строки

```
private static void writeTrie(Node x)
{  // Запись trie-дерева в виде битовой строки.
   if (x.isLeaf())
   {
      BinaryStdOut.write(true);
      BinaryStdOut.write(x.ch);
      return;
   }
```

```
    BinaryStdOut.write(false);
    writeTrie(x.left);
    writeTrie(x.right);
}
```

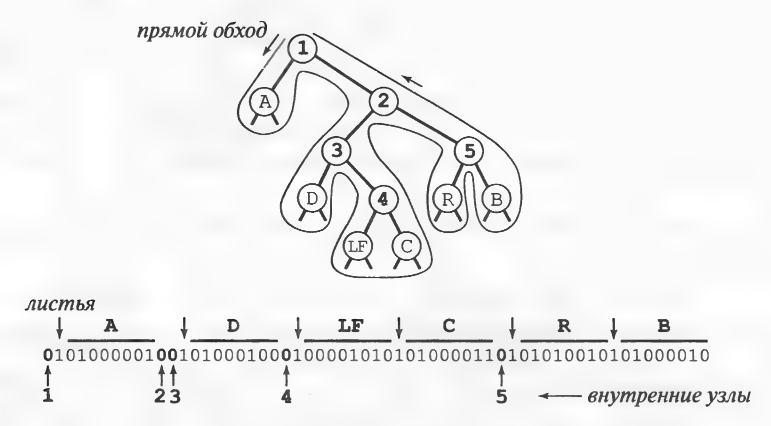


Рис. 5.5.15. Кодирование trie-дерева в виде битового потока с помощью прямого обхода

Соответствующий метод `readTrie()`, приведенный в листинге 5.5.13, восстанавливает trie-дерево из битовой строки. Он читает один бит и определяет тип следующего узла: если это лист (бит равен `1`), читается следующий символ и создается лист, а если это внутренний узел (бит равен `0`), создается внутренний узел, а затем (рекурсивно) строятся его левое и правое поддеревья. *Обязательно разберитесь в этих методах*: они не так просты, как кажется на первый взгляд.

Листинг 5.5.13. Восстановление trie-дерева из представления битовой строкой прямого обхода

```
private static Node readTrie()
{
   if (BinaryStdIn.readBoolean())
      return new Node(BinaryStdIn.readChar(), 0, null, null);
   return new Node('\0', 0, readTrie(), readTrie());
}
```

Реализация сжатия Хаффмана

Вместе с только что рассмотренными методами `buildCode()`, `buildTrie()`, `readTrie()` и `writeTrie()` (и методом `expand()`, с которым мы познакомились несколько раньше) алгоритм 5.10, приведенный в листинге 5.5.14, представляет собой полную реализацию сжатия Хаффмана. Теперь можно расширить краткое содержание метода и описать кодирование потока 8-битовых значений `char` в виде последовательности следующих шагов.

- Чтение входных данных.
- Подсчет количества повторений каждого значения `char` во входных данных.
- Построение trie-дерева для кодирования Хаффмана, соответствующего этим счетчикам.

- Построение соответствующей таблицы кодовых последовательностей, которая ставит в соответствие битовую строку каждому значению `char` из входных данных.
- Запись trie-дерева в виде битовой строки.
- Запись количества символов во входных данных в виде битовой строки.
- Использование построенной таблицы для записи кодовых последовательностей для каждого входного символа.

Для распаковки битового потока, упакованного таким образом, нужно выполнить следующие шаги.

- Чтение trie-дерева (записанного в начале битового потока).
- Чтение счетчика декодируемых символов.
- Использование trie-дерева для декодирования битового потока.

Листинг 5.5.14. Сжатие Хаффмана

```
public class Huffman
{
   private static int R = 256;   // Алфавит ASCII
   // Внутренний класс Node см. в листинге 5.5.7.
   // Вспомогательные методы и expand() приведены в тексте.

   public static void compress()
   {
      // Чтение входных данных.
      String s = BinaryStdIn.readString();
      char[] input = s.toCharArray();

      // Подсчет частот символов.
      int[] freq = new int[R];
      for (int i = 0; i < input.length; i++)
         freq[input[i]]++;

      // Построение trie-дерева для кода Хаффмана.
      Node root = buildTrie(freq);

      // Построение таблицы кодировки (рекурсивное).
      String[] st = new String[R];
      buildCode(st, root, "");

      // Вывод trie-дерева для декодирования (рекурсивное).
      writeTrie(root);

      // Вывод количества символов.
      BinaryStdOut.write(input.length);

      // Кодирование входных данных с помощью кода Хаффмана.
      for (int i = 0; i < input.length; i++)
      {
         String code = st[input[i]];
         for (int j = 0; j < code.length(); j++)
         if (code.charAt(j) == '1')
             BinaryStdOut.write(true);
         else BinaryStdOut.write(false);
      }
      BinaryStdOut.close();
   }
```

Эта реализация кодирования Хаффмана (рис. 5.5.16) строит явное trie-дерево кодирования, используя различные вспомогательные методы, которые представлены и объяснены на последних нескольких страницах текста.

Сжатие Хаффмана, которое выполняется с помощью четырех рекурсивных методов обработки trie-деревьев и семи только что описанных шагов, является одним из наиболее сложных алгоритмов из рассматриваемых нами, но, в силу своей эффективности, это и один из наиболее употребительных алгоритмов.

Одной из причин популярности сжатия Хаффмана является его эффективность для различных видов файлов, а не только для текстов на естественных языках. Мы тщательно закодировали этот метод, чтобы он правильно работал для любых 8-битовых символов. То есть он применим вообще к любым байтовым потокам. На рис. 5.5.17 приведено несколько примеров для видов файлов, которые были рассмотрены ранее в данном разделе. Эти примеры показывают, что сжатие Хаффмана вполне успешно конкурирует и с кодированием фиксированной длины, и с кодированием по длинам серий, хотя эти методы были специально созданы для хорошей производительности на определенных типах файлов. Полезно разобраться в причинах хорошей производительности кодирования Хаффмана в подобных областях. В случае генетических данных сжатие Хаффмана, по сути, находит 2-битовый код, т.к. все четыре буквы появляются с примерно одинаковой частотой, и в силу сбалансированности дерева Хаффмана каждой букве назначается 2-битовая кодовая последовательность. В случае кодирования по длинам серий зачастую последовательности `00000000` и `11111111` встречаются чаще других и поэтому кодируются двумя или тремя битами, что приводит к существенному сжатию.

Интересная альтернатива сжатию Хаффмана была разработана в конце 1970-х и в начале 1980-х годов А. Лемпелем (A. Lempel), Дж. Зивом (J. Ziv) и Т. Велчем (T. Welch). Этот способ стал одним из наиболее широко применяемых методов сжатия, поскольку он легко реализуется и хорошо работает для многих различных видов файлов.

Базовый принцип этого способа обратен базовому принципу кодирования Хаффмана. Вместо использования таблицы кодовых последовательностей *переменной* длины для фрагментов входных данных *фиксированной* длины в нем используется таблица кодовых последовательностей *фиксированной* длины для фрагментов входных данных *переменной* длины. Неожиданным дополнительным свойством этого метода является то, что в отличие от кодирования Хаффмана, в нем *не нужно кодировать таблицу*.

LZW-сжатие

Для наглядности мы рассмотрим пример сжатия, где выполняется чтение входных данных как потока 7-битовых символов ASCII и запись выходных данных как потока 8-битовых байтов. (На практике для этих параметров обычно используются большие значения — в наших реализациях это 8-битовый ввод и 12-битовый вывод.) Входные байты мы будем называть *символами*, последовательности входных байтов — *строками*, а выходные байты — *кодовыми последовательностями*, хотя в других контекстах эти термины имеют несколько другой смысл. Алгоритм LZW-сжатия основан на использовании таблицы имен, которая связывает строковые ключи со значениями кодовых последовательностей (фиксированной длины). Вначале в таблицу имен заносятся 128 возможных строковых ключей из отдельных символов, и они связываются с 8-битовыми кодовыми последовательностями, полученными приписыванием нулевого бита в начало 7-битового значения, определяющего каждый символ.

Тестовый пример (96 битов)

```
% more abra.txt
ABRACADABRA!

% java Huffman - < abra.txt | java BinaryDump 60
010100000100101000100010000101010100001101010100101010000100
000000000000000000000000000110001111100101101000111110010100
120 bits  ←— коэффициент сжатия 120/96 = 125%, из-за присутствия 59 битов
             для trie-дерева и 32 битов для счетчика
```

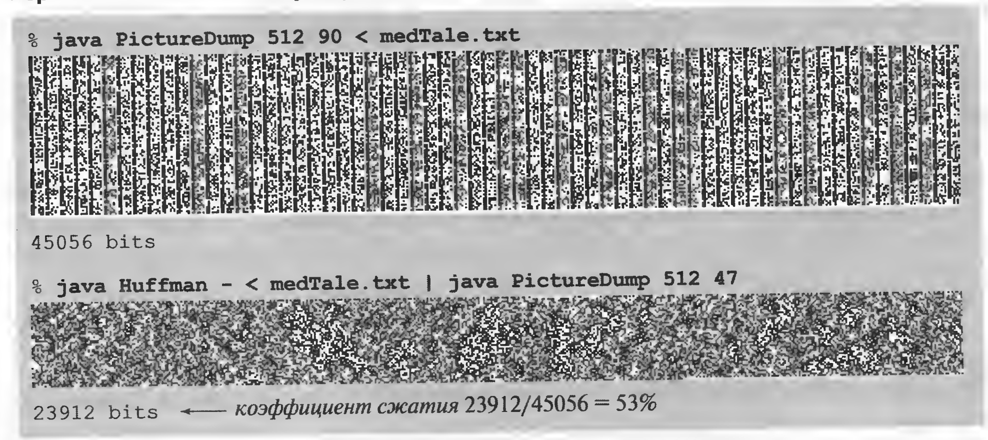
Пример из текста (408 битов)

```
% more tinytinyTale.txt
it was the best of times it was the worst of times

% java Huffman - < tinytinyTale.txt | java BinaryDump 64
0001011001010101110111101101111100100000001011100110010111001001
0000101010110001010110100100010110011010110100001011011011011000
0110111010000000000000000000000000000110011101111101001011011100
0111111001000011010110001001110100111100001111101111101000100011
0111110100101101110001111110010000100100011101001001110100111100
00111110111110100001001010100000
352 bits  ←— коэффициент сжатия 352/408 = 86%, даже с 137 битами
             для trie-дерева и 32 битами для счетчика

% java Huffman - < tinytinyTale.txt | java Huffman +
it was the best of times it was the worst of times
```

Первая глава “Повести о двух городах”



Весь текст “Повести о двух городах”

```
% java BinaryDump 0 < tale.txt
5812552 bits

% java Huffman - < tale.txt > tale.txt.huf
% java BinaryDump 0 < tale.txt.huf
3043928 bits  ←— коэффициент сжатия 3043928/5812552 = 52%
```

Рис. 5.5.16. Упаковка и распаковка битовых потоков с помощью кодирования Хаффмана

Вирус (50 000 битов)

```
% java Genome  - < genomeVirus.txt | java PictureDump 512 25
```

```
12536 bits
```

```
% java Huffman - < genomeVirus.txt | java PictureDump 512 25
```

```
12576 bits
```

← *Сжатие Хаффмана требует лишь на 40 битов больше, чем специальный 2-битовый код*

Растровое изображение (1536 битов)

```
% java RunLength - < q32x48.bin | java BinaryDump 0
1144 bits

% java Huffman   - < q32x48.bin | java BinaryDump 0
816 bits
```

← *Сжатие Хаффмана использует на 29% меньше битов, чем специальный метод*

Растровое изображение с более высоким разрешением (6144 бита)

```
% java RunLength - < q64x96.bin | java BinaryDump 0
2296 bits

% java Huffman   - < q64x96.bin | java BinaryDump 0
2032 bits
```

← *Для повышенного разрешения разрыв увеличился до 11%*

Рис. 5.5.17. *Упаковка и распаковка геномных данных и растровых изображений с помощью кодирования Хаффмана*

Для экономии места и ясности при записи кодовых последовательностей мы будем использовать шестнадцатеричные числа. Поэтому `41` будет кодовой последовательностью для ASCII-символа `A`, `52` — для `R` и т.д. Кодовая последовательность `80` будет означать конец файла. Остальные кодовые последовательности (с `81` по `FF`) мы будем назначать различным подстрокам из входных данных, начиная с `81` и далее для каждого нового добавляемого ключа. Для выполнения сжатия выполняются следующие шаги, пока есть непросмотренные входные символы.

- В таблице имен находится самая длинная строка `s`, которая является префиксом непросмотренных входных данных.
- Записывается 8-битовое значение (кодовая последовательность), связанная с `s`.
- Позиция просмотра во входных данных сдвигается за `s`.
- В таблице имен находится следующее кодовое значение, связанное с `s + c` (конкатенация), где `c` — следующий входной символ.

В последнем из этих шагов мы заглядываем в следующий входной символ, чтобы создать еще один элемент словаря, поэтому мы называем этот символ `c` *авангардным* (lookahead) символом. Пока мы будем просто прекращать добавлять элементы в таблицу имен, когда закончатся значения кодовых последовательностей (после присваивания какой-то строке значения `FF`), а потом рассмотрим и другие стратегии.

Пример LZW-сжатия

На рис. 5.5.18 подробно изображен процесс LZW-упаковки для примера входных данных ABRACADABRABRABRA. Для первых семи символов самый длинный префикс соответствует лишь одному символу, поэтому выводится кодовая последовательность, связанная с этим символом, и кодовые последовательности с 81 по 87 связываются с двухсимвольными строками. Потом обнаруживаются префиксы AB (выводится 81, а строка ABR заносится в таблицу), RA (выводится 83, а RAC заносится в таблицу), BR (выводится 82, а BRA заносится в таблицу) и ABR (выводится 88, а ABRA заносится в таблицу). После этого остается последний символ A (выводится его кодовая последовательность 41).

Вход	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A	*EOF*
Соответствие	A	B	R	A	C	A	D	A B		R A		B R		A B R			A	↓
Выход	41	42	52	41	43	41	44	81		83		82		88			41	80

AB81	AB	AB	AB	AB	AB	AB	AB	AB	AB	AB
	BR82	BR	BR	BR	BR	BR	BR	BR	BR	BR
		RA83	RA	RA	RA	RA	RA	RA	RA	RA
			AC84	AC	AC	AC	AC	AC	AC	AC
				CA85	CA	CA	CA	CA	CA	CA
					AD86	AD	AD	AD	AD	AD
						DA87	DA	DA	DA	DA
							ABR88	ABR	ABR	ABR
								RAB89	RAB	RAB
									BRA8A	BRA
										ABRA8B

входная подстрока ↑ (AB81); *кодовая последовательность* ↑ (82); *авангардный символ* ↑ (A в CA85)

Кодовая таблица

Ключ	*Значение*
AB	81
BR	82
RA	83
AC	84
CA	85
AD	86
DA	87
ABR	88
RAB	89
BRA	8A
ABRA	8B

Рис. 5.5.18. *LZW-упаковка для строки* ABRACADABRABRABRA

На входе было 17 ASCII-символов из 7 битов каждый, общим объемом 119 битов; на выходе — 12 кодовых последовательностей по 8 битов каждая, общим объемом 96 битов. Даже для такого крошечного примера получился коэффициент сжатия 82%.

Представление LZW-дерева

Для LZW-сжатия необходимы две операции с таблицей имен.

- Поиск наиболее длинного префикса из входных данных с помощью ключа таблицы имен.
- Добавление элемента, который связывает следующую кодовую последовательность с ключом, сформированным добавлением авангардного символа к этому ключу.

Наши структуры данных trie-деревьев из раздела 5.2 просто специально приспособлены для выполнения этих операций. Представление trie-дерева для нашего примера приведено на рис. 5.5.19. Чтобы найти соответствие с самым длинным префиксом, мы проходим по trie-дереву от корня, сравнивая метки в узлах с входными символами; чтобы добавить новую кодовую последовательность, мы присоединяем новый узел, помеченный следующим кодовым словом и авангардным символом, к узлу, где

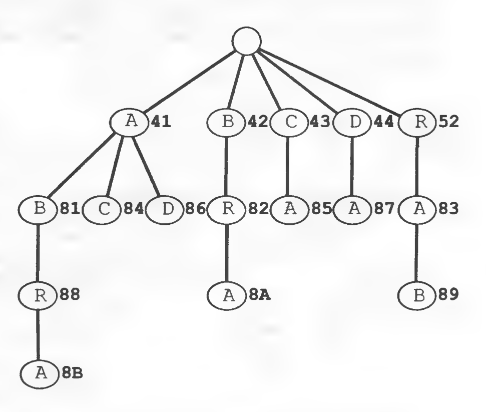


Рис. 5.5.19. *Представление кодовой таблицы LZW в виде trie-дерева*

поиск был завершен. На практике для экономии памяти используется trie-дерево тернарного поиска (ТТП), описанное в разделе 5.2. Контраст с использованием trie-деревьев в кодировании Хаффмана на самом деле несущественен: при кодировании Хаффмана trie-деревья применяются потому, что ни один префикс кодовой последовательности не может быть другой кодовой последовательностью, в а в LZW trie-деревья применяются потому, что *каждый* префикс ключа из входной подстроки также является ключом.

LZW-распаковка

Входными данными для LZW-распаковки в нашем примере является последовательность 8-битовых кодовых последовательностей, а выходными данными — строка 7-битовых символов ASCII. Для реализации распаковки используется таблица имен, которая ставит в соответствие строки символов значениям кодовых последовательностей (обращение таблицы, применяемой для упаковки). Элементы таблицы от `00` по `FF` заполняются односимвольными строками, по одной для каждого символа ASCII, первой неназначенной кодовой последовательности присваивается значение `81` (`80` резервируется для конца файла), в текущую строку `val` заносится односимвольная строка, содержащая первый символ, а затем выполняются следующие шаги, пока не встретится кодовая последовательность `80` (конец файла).

- Записывается текущая строка `val`.
- Из входных данных читается кодовая последовательность `x`.
- В `s` заносится значение, связанное с `x` в таблице имен.
- Значение следующей неназначенной кодовой последовательности связывается с элементом `val + c` в таблице имен, где `c` — первый символ `s`.
- В текущую строку `val` заносится `s`.

Этот процесс (рис. 5.5.20) сложнее упаковки из-за авангардного символа: необходимо читать следующую кодовую последовательность, чтобы получить первый символ в строке, связанной с ним, а это рассинхронизирует процесс на один шаг. Для первых семи кодовых последовательностей мы просто ищем и записываем соответствующий символ, а затем заглядываем на один символ вперед и добавляем, как и раньше, в таблицу имен двухсимвольный элемент. Затем мы читаем последовательность `81` (записываем `AB` и заносим в таблицу `ABR`), `83` (записываем `RA` и заносим в таблицу `RAB`), `82` (записываем `BR` и заносим в таблицу `BRA`) и `88` (записываем `ABR` и заносим в таблицу `ABRA`), после чего остается `41`.

Вход	41	42	52	41	43	41	44	81	83	82	88	41	80
Выход	A	B	R	A	C	A	D	A B	R A	B R	A B R	A	

81AB, 82BR, 83RA, 84AC, 85CA, 86AD, 87DA, 88ABR, 89RAB, 8AABRA, 8B ABRA

кодовая последовательность

входная подстрока

Инвертированная кодовая таблица

Ключ	*Значение*
81	AB
82	BR
83	RA
84	AC
85	CA
86	AD
87	DA
88	ABR
89	RAB
8A	BRA
8B	ABRA

***Рис. 5.5.20.** LZW-распаковка для последовательности*
41 42 52 41 43 41 44 81 83 82 88 41 80

В завершение мы читаем символ конца файла 80 (и записываем A). В конце процесса на выходе получились, как и ожидалось, исходные входные данные, а также построена та же кодовая таблица, что и на этапе упаковки, но с обменом ролей ключей и значений. Для такой таблицы можно использовать простое представление массивом строк, индексированным кодовыми последовательностями.

Особая ситуация

В вышеописанном процессе имеется одна тонкая ошибка, которая обнаруживается студентами (да и опытными программистами!) только после разработки реализации, основанной на приведенном описании. Эта проблема, показанная на рис. 5.5.21, состоит в том, что процесс заглядывания вперед может опередить сам себя на один символ. В нашем примере входная строка ABABABA упаковывается в пять выходных кодовых последовательностей 41 42 81 83 80, как показано в верхней части рисунка. При распаковке мы читаем кодовую последовательность 41, выводим A, читаем 42, чтобы получить авангардный символ, заносим AB как элемент таблицы 81, выводим символ B, связанный с последовательностью 42, читаем 81, чтобы получить авангардный символ, заносим BA как элемент таблицы 82 и выводим строку AB, связанную с последовательностью 81. Пока все нормально. Но когда мы читаем кодовую последовательность 83, чтобы получить авангардный символ, алгоритм буксует, т.к. мы читаем эту последовательность для того, чтобы заполнить элемент таблицы 83! К счастью, нетрудно организовать проверку такой ситуации (она встречается как раз тогда, когда кодовая последовательность совпадает с заполняемым элементом таблицы) и исправить ее: авангардный символ должен быть первым символом в этом элементе таблицы, поскольку это будет следующий выводимый символ. В данном примере эта логика подсказывает, что авангардный символ должен быть A — первый символ в строке ABA. Поэтому и следующая выходная строка, и элемент таблицы 83 должны быть равны ABA.

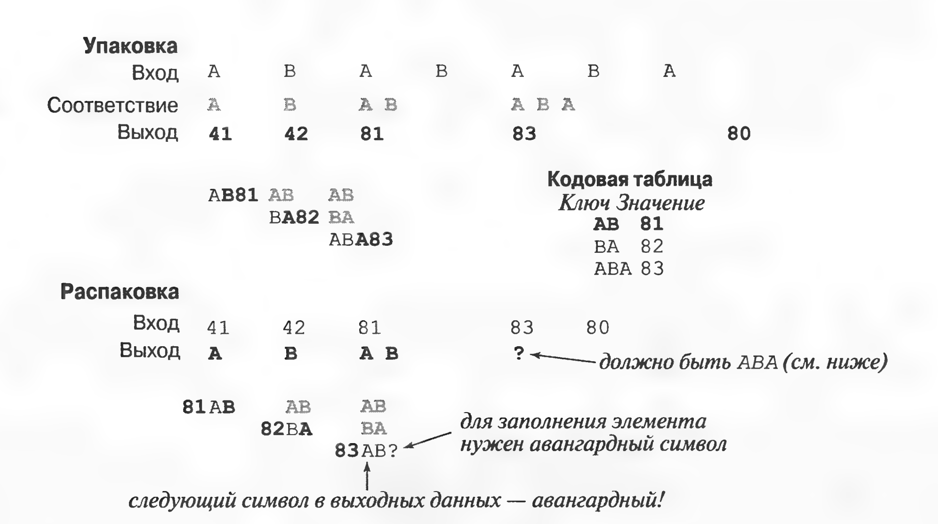


Рис. 5.5.21. LZW-распаковка: особая ситуация

Реализация

С учетом приведенных описаний реализация LZW-сжатия уже не представляет труда — см. алгоритм 5.11 в листинге 5.5.15 и реализацию метода `expand()` в листинге 5.5.16. Эти реализации принимают в качестве входных данных 8-битовые байты (т.е. можно сжимать любой файл, а не только строки) и выдают 12-битовые кодовые последовательности (можно получить гораздо лучшее сжатие с помощью гораздо большего словаря). Эти значения указаны в (финальных) переменных экземпляров `R`, `L` и `W` в коде алгоритма. Для кодовой таблицы в методе `compress()` используется trie-дерево тернарного поиска (см. раздел 5.2), которое позволяет поддерживать эффективные реализации метода `longestPrefixOf()`, и массив строк для инвертированной кодовой таблицы в методе `expand()`. После выбора этих структур код методов `compress()` и `expand()` сводится просто к построчной записи описаний из текста. Эти методы очень эффективны даже в том виде, в каком они записаны. Но для некоторых файлов их можно еще усовершенствовать, очищая таблицу кодовых последовательностей и начиная сначала каждый раз, когда использованы все значения кодовых последовательностей. Эти усовершенствования, а также эксперименты по оценке их эффективности, описаны в упражнениях в конце данного раздела.

Листинг 5.5.15. Алгоритм 5.11. LZW-упаковка

```
public class LZW
{
   private static final int R = 256;   // Количество входных символов
   private static final int L = 4096;  // Количество кодовых
                                       //    последовательностей = 2^12
   private static final int W = 12;    // Ширина кодовой последовательности

   public static void compress()
   {
      String input = BinaryStdIn.readString();
      TST<Integer> st = new TST<Integer>();
      for (int i = 0; i < R; i++)
         st.put("" + (char) i, i);
      int code = R+1;                  // R — кодовая последовательность для EOF.
      while (input.length() > 0)
      {
         String s = st.longestPrefixOf(input); // Поиск максимального
                                               //   соответствия префикса.
         BinaryStdOut.write(st.get(s), W);     // Вывод кодировки для s.
         int t = s.length();
         if (t < input.length() && code < L)   // Добавление s в таблицу имен.
            st.put(input.substring(0, t + 1), code++);
         input = input.substring(t);           // Просмотр входных данных после s.
      }
      BinaryStdOut.write(R, W);                // Запись EOF.
      BinaryStdOut.close();
   }
   public static void expand()
   // См. листинг 5.5.16.
}
```

Эта реализация упаковки данных методом Лемпеля-Зива-Велча использует 8-битовые входные байты и 12-битовые кодовые последовательности и пригодна для

файлов произвольного размера. Кодовые последовательности для небольшого примера аналогичны рассмотренным в тексте: последовательности из одного символа содержат старший 0, остальные начинаются с 100.

```
% more abraLZW.txt
ABRACADABRABRABRA

% java LZW - < abraLZW.txt | java HexDump 20
04 10 42 05 20 41 04 30 41 04 41 01 10 31 02 10 80 41 10 00
160 bits
```

Листинг 5.5.16. Алгоритм 5.11 (продолжение). LZW-распаковка

```
public static void  expand()
{
  String[] st = new String[L];
  int i; // Следующее доступное значение кодовой последовательности
  for (i = 0; i < R; i++)           // Инициализация таблицы для символов.
    st[i] = "" + (char) i;
  st[i++] = " ";                    // (не используется) Заглядывание
                                    //   вперед, EOF ли это
  int codeword = BinaryStdIn.readInt(W);
  String val = st[codeword];
  while (true)
  {
    BinaryStdOut.write(val);        // Запись текущей подстроки.
    codeword = BinaryStdIn.readInt(W);
    if (codeword == R) break;
    String s = st[codeword];        // Получение следующей кодовой
                                    //   последовательности.
    if (i == codeword)              // Если заглядывание выполняется неверно,
      s = val + val.charAt(0);      //   кодовая последовательность
                                    //   формируется из последней.
    if (i < L)
      st[i++] = val + s.charAt(0);  // Добавление нового элемента
                                    //   в кодовую таблицу.
    val = s;                        // Изменение текущей кодовой
                                    //   последовательности.
  }
  BinaryStdOut.close();
}
```

Эта реализация распаковки для алгоритма Лемпеля-Зива-Велча несколько сложнее, чем упаковка, из-за необходимости выбирать авангардный символ из следующей кодовой последовательности и из-за особой ситуации, когда заглядывание вперед выполняется неправильно (см. текст).

```
% java LZW - < abraLZW.txt | java LZW +
ABRACADABRABRABRA
% more ababLZW.txt
ABABABA
% java LZW - < ababLZW.txt | java LZW +
ABABABA
```

Как обычно, стоит не пожалеть времени на тщательное изучение примеров работы LZW-сжатия, приведенных на рис. 5.5.22. На протяжении нескольких десятилетий с момента его изобретения он зарекомендовал себя как универсальный и эффективный метод сжатия данных.

Вирус (50 000 битов)

```
% java Genome - < genomeVirus.txt | java PictureDump 512 25
```

```
12536 bits
```

```
% java LZW - < genomeVirus.txt | java PictureDump 512 36
```

```
18232 bits
```
← *не так хорош, как 2-битовый код, т.к. повторения в данных встречаются нечасто*

Растровое изображение (6144 бита)

```
% java RunLength - < q64x96.bin | java BinaryDump 0
2296 bits

% java LZW - < q64x96.bin | java BinaryDump 0
2824 bits
```
← *не так хорош, как кодирование по длинам серий, т.к. размер файла слишком мал*

Весь текст "Повести о двух городах" (5 812 552 бита)

```
% java BinaryDump 0 < tale.txt
5812552 bits

% java Huffman - < tale.txt | java BinaryDump 0
3043928 bits

% java LZW - < tale.txt | java BinaryDump 0
2667952 bits
```
← *коэффициент сжатия* 2667952 / 5812552 = 46% *(лучший из полученных нами)*

***Рис. 5.5.22.** Упаковка и распаковка различных файлов с помощью 12-битового LZW-кодирования*

Вопросы и ответы

Вопрос. **Зачем нужны классы `BinaryStdIn` и `BinaryStdOut`?**

Ответ. Это компромисс между эффективностью и удобством. `StdIn` обрабатывает сразу по 8 битов, а `BinaryStdIn` обрабатывает каждый бит. Большинство приложений ориентировано на байтовые потоки, но сжатие данных является исключением.

Вопрос. **Зачем нужен метод `close()`?**

Ответ. Это требование объясняется тем, что стандартный вывод в реальности является байтовым потоком, поэтому классу `BinaryStdOut` нужно знать, когда записывать последний байт.

Вопрос. **Можно ли совместно использовать StdIn и BinaryStdIn?**

Ответ. Лучше не надо. Из-за особенностей системы и реализации невозможно предсказать, что при этом получится. В наших реализациях будет сгенерировано исключение. А вот StdOut и BinaryStdOut можно смешивать без проблем (и так сделано в нашем коде).

Вопрос. **Почему в коде Huffman класс Node объявлен как static?**

Ответ. Наши алгоритмы сжатия данных организованы в виде коллекций статических методов, а не реализаций типов данных.

Вопрос. **Можно ли, по крайней мере, гарантировать, что алгоритм сжатия не увеличит длину битового потока?**

Ответ. Можно просто скопировать поток из ввода в вывод, но тогда необходимо указать, что не нужно использовать стандартную схему упаковки/распаковки. В коммерческих реализациях иногда такая гарантия дается, но это весьма слабо похоже на универсальное сжатие. Обычные алгоритмы сжатия даже не преодолевают второй шаг нашего первого доказательства утверждения У: лишь немногие алгоритмы могут сжать битовую строку, полученную этим же алгоритмом.

Упражнения

5.5.1. В табл. 5.5.2 приведены четыре кода переменной длины. Какие из них являются беспрефиксными? Уникально декодируемыми? Для уникально декодируемых распакуйте строку 1000000000000.

Таблица 5.5.2. Таблица к упражнению 5.5.1

Символ	Код 1	Код 2	Код 3	Код 4
A	0	0	1	1
B	100	1	01	01
C	10	00	001	001
D	11	11	0001	000

5.5.2. Приведите пример уникально декодируемого кода, который не является беспрефиксным.

Ответ: любой беспрефиксный код является уникально декодируемым.

5.5.3. Приведите пример уникально декодируемого кода, который не является бепрефиксным или бессуфиксным.

Ответ: {0011, 011, 11, 1110} или {01, 10, 011, 110} .

5.5.4. Являются ли коды {01, 1001, 1011, 111, 1110} и {01, 1001, 1011, 111, 1110} уникально декодируемыми? Если нет, приведите строку с двумя вариантами кодировки.

5.5.5. Упакуйте файл q128x192.bin с сайта книги программой RunLength. Сколько битов содержит сжатый файл?

5.5.6. Сколько битов требуется для кодирования N копий символа a (в виде функции от N)? А для N копий подстроки abc?

5.5.7. Приведите результат кодирования строк a, aa, aaa, aaaa, ... (строк, состоящих из N букв a) с помощью кодирования по длинам серий, кодирования Хаффмана и LZW-кодирования. Чему равен коэффициент сжатия в виде функции от N?

5.5.8. Приведите результат кодирования строк ab, abab, ababab, abababab, ... (строк, состоящих из N повторений ab) с помощью кодирования по длинам серий, кодирования Хаффмана и LZW-кодирования. Чему равен коэффициент сжатия в виде функции от N?

5.5.9. Приведите оценку коэффициента сжатия, получаемого кодированием по длинам серий, Хаффмана и LZW, для *случайных* ASCII строк длиной N (все символы могут независимо появиться в любой позиции с одинаковой вероятностью).

5.5.10. В стиле рисунков, приведенных в тексте, продемонстрируйте процесс построения для дерева кодирования Хаффмана при использовании программы Huffman для строки "it was the age of foolishness". Сколько битов потребуется для сжатого битового потока?

5.5.11. Каким будет код Хаффмана для строки, все символы которой взяты из двухсимвольного алфавита? Приведите пример с максимальным количеством битов, которые может использовать код Хаффмана для N-символьной строки, символы которой взяты из двухсимвольного алфавита.

5.5.12. Пусть вероятности появления всех символов равны отрицательным степеням 2. Опишите код Хаффмана.

5.5.13. Пусть вероятности появления всех символов одинаковы. Опишите код Хаффмана.

5.5.14. Пусть вероятности появления всех кодируемых символов различны. Уникально ли дерево кодирования Хаффмана?

5.5.15. Кодирование Хаффмана можно естественно расширить для кодирования 2-битовых символов (с помощью 4-частных деревьев). Каково основное достоинство и основной недостаток этого способа?

5.5.16. Каковы результаты LZW-кодирования следующих входных данных?

а) T O B E O R N O T T O B E

б) Y A B B A D A B B A D A B B A D O O

в) A A A A A A A A A A A A A A A A A A A A A

5.5.17. Опишите особую ситуацию в LZW-кодировании.

Решение: когда встречается строка cScSc, где c — символ, а S — строка, cS уже находится в словаре, а cSc — нет.

5.5.18. Пусть F_k — k-е число Фибоначчи. Рассмотрим N символов, где k-й символ встречается в тексте с частотой F_k. Опишите соответствующий код Хаффмана, учитывая, что $F_1 + F_2 + ... + F_N = F_{N+2} - 1$. *Подсказка*: самая длинная кодовая последовательность имеет длину $N-1$.

5.5.19. Покажите, что для заданного множества N символов существуют, по крайней мере, 2^{N-1} различных кодов Хаффмана.

5.5.20. Приведите код Хаффмана, где частота появления нулевых битов во много раз превышает частоту появления единичных битов.

Ответ: если символ `A` встречается в миллион раз чаще, чем символ `B`, то кодовая последовательность для `A` будет `0`, а для `B` — `1`.

5.5.21. Докажите, что две самых длинных кодовых последовательности в коде Хаффмана имеют одинаковую длину.

5.5.22. Докажите следующее утверждение относительно кодов Хаффмана: если частота символа i заметно больше частоты символа j, то длина кодовой последовательности для символа i меньше или равна длине кодовой последовательности для символа j.

5.5.23. Что получится, если разбить результат кодирования Хаффмана на 5-битовые символы и применить сжатие Хаффмана к этой строке?

5.5.24. В стиле рисунков, приведенных в тексте, покажите trie-дерево кодирования и процессы упаковки/распаковки при работе программы `LZW` со строкой

```
it was the best of times it was the worst of times
```

Творческие задачи

5.5.25. *Код с фиксированной длиной.* Реализуйте класс `RLE`, который выполняет кодирование фиксированной длины для сжатия потоков ASCII-байтов, содержащих относительно немного различных символов, и включает код в закодированный битовый поток. Добавьте в метод `compress()` код формирования строки `alpha`, которая содержит все различные символы из сообщения, и используйте ее для получения объекта `Alphabet` для метода `compress()`. Записывайте строку `alpha` (8-битовая кодировка плюс длина) в начало битового потока, и добавьте в метод `expand()` чтение алфавита перед распаковкой.

5.5.26. *Перестроение словаря в методе LZW.* Добавьте в программу `LZW` очистку словаря и заполнение его снова, когда словарь заполнится. Этот подход рекомендуется в некоторых приложениях, поскольку он лучше приспосабливается к изменениям общего характера входных данных.

5.5.27. *Длинные повторы.* Оцените коэффициент сжатия, который получается при кодировании по длинам серий, Хаффмана и LZW для строки длиной $2N$, сформированной конкатенацией *двух копий* случайной ASCII-строки длиной N (см. упражнение 5.5.9), при разумных (по вашему мнению) предположениях.

Глава 6

Контекст

В современном мире компьютерные устройства можно обнаружить везде. В последние несколько десятилетий мы перешли из мира, в котором вычислительные устройства были практически неизвестны, в мир, где миллиарды людей пользуются ими регулярно. Даже современные мобильные телефоны на порядки мощнее суперкомпьютеров, к которым имели доступ только немногие избранные всего лишь 30 лет назад. Но многие из алгоритмов, которые позволяют эффективно работать этим устройствам — те, которые мы изучили в данной книге. Почему? *Потому что выживают лучшие.* Масштабируемые (линейные и линейно-логарифмические) алгоритмы играют центральную роль в этих процессах и подтверждают тезис о важности эффективности алгоритмов. Исследователи в 1960–1970-х годах создали базовую инфраструктуру, которая основана на подобных алгоритмах. Они понимали, что масштабируемые алгоритмы — ключ к будущему, и разработки последних нескольких десятилетий подтвердили это понимание. Теперь, когда эта инфраструктура уже создана, люди начинают *использовать* ее для разнообразных целей. Как когда-то заметил Б. Чезель (B. Chazelle), XX век был веком уравнений, а XXI — это век *алгоритмов*.

Наше знакомство с фундаментальными алгоритмами в этой книге представляет собой лишь отправную точку. Скоро наступит (если еще не наступил) день, когда в колледжах появится целая специальность по изучению алгоритмов. В коммерческих приложениях, научном программировании, инженерных расчетах, исследовании операций и в других многочисленных областях, которые долго даже просто перечислять, эффективные алгоритмы позволяют решать задачи, которые иначе решить невозможно. В данной книге мы старались рассматривать *важные* и *полезные* алгоритмы. И, чтобы еще раз подчеркнуть такую направленность, в этой главе мы рассмотрим примеры, которые демонстрируют роль изученных нами алгоритмов (и наш способ изучения алгоритмов) в нескольких более сложных контекстах. Чтобы обозначить область применимости этих алгоритмов, вначале мы предоставим очень краткое описание нескольких важных областей применения. А чтобы обозначить глубину, позже мы подробно рассмотрим которые характерные примеры, а заодно и введение в теорию алгоритмов. В обоих случаях этот краткий обзор в конце объемной книги может быть лишь демонстрационным, но не всеобъемлющим. Для каждой упомянутой нами предметной области имеются десятки других, с не меньшей области применимости; для каждого положения, описанного в применении, имеются десятки других, не менее важных; и для каждого рассмотренного нами подробного примера существуют сотни, если не тысячи, других, не менее внушительных.

Коммерческие приложения

Распространение Интернета высветило центральную роль алгоритмов в *коммерческих приложениях*. Все регулярно используемые вами приложения выигрывают от применения изученных здесь классических алгоритмов.

- инфраструктура (операционные системы, базы данных, коммуникации);
- приложения (электронная почта, обработка документов, цифровая фотография);
- издательское дело (книги, журналы, веб-контент);
- сети (беспроводные сети, социальные сети, Интернет);
- обработка транзакций (банки, розничная продажа, веб-поиск).

В качестве очень заметного примера мы рассмотрим в этой главе *В-деревья* — структуру данных, которая была разработана для компьютерной обработки данных еще в 1960-х годах, но до сих пор применяется в современных системах баз данных. Кроме того, мы рассмотрим применение *суффиксных массивов* для индексации текста.

Научные вычисления

Со времени разработки фон Нейманом сортировки слиянием в 1950 г. алгоритмы играют центральную роль в *научных вычислениях*. Современные ученые работают с огромными объемами данных и используют как математические, так и вычислительные модели для выяснения сути окружающего нас мира:

- математические вычисления (полиномы, матрицы, дифференциальные уравнения);
- обработка данных (результаты экспериментов и наблюдений, особенно в геномике);
- вычислительные модели и компьютерное моделирование.

Во всех этих направлениях может потребоваться сложная и интенсивная обработка огромных объемов данных. В качестве подробного примера из области научных вычислений мы рассмотрим в данной главе классический пример *событийного моделирования*. Идея состоит в использовании модели сложной реальной системы, отражающей происходящие различные изменения. На этом базовом подходе основано огромное количество приложений. Мы также рассмотрим фундаментальную задачу обработки данных в вычислительной геномике.

Инженерные расчеты

Практически по определению, современное машиностроение и приборостроение основано на технологии. Современные технологии немыслимы без компьютеров, поэтому алгоритмы играют центральную роль в таких областях:

- математические вычисления и обработка данных;
- компьютерное конструирование и производство;
- проектирование на основе алгоритмов (сети, системы управления);
- работа с изображениями и другие медицинские системы.

Инженеры и ученые во многом используют те же средства и способы. Например, ученые разрабатывают вычислительные модели и компьютерное моделирование, чтобы разобраться в явлениях реального мира, а инженеры создают вычислительные модели для проектирования и создания своих творений и управления ими.

Исследование операций (ИО)

Исследователи и те, кто применяет ИО на практике, разрабатывают и применяют математические модели для решения различных задач, в числе которых:

- планирование;
- принятие решений;
- распределение ресурсов.

Задача поиска кратчайших путей из раздела 4.4 — классическая задача ИО. Мы еще вернемся к ней и рассмотрим задачу *максимального потока*, продемонстрируем важность *сведения* и рассмотрим выводы из общих моделей решения задач, в частности, из модели *линейного программирования*, которая является центральной в ИО.

Алгоритмы играют важную роль в многочисленных более мелких областях компьютерных наук с соответствующими приложениями, среди которых, в частности:

- вычислительная геометрия;
- криптография;
- базы данных;
- языки и системы программирования;
- искусственный интеллект.

В каждой такой области важную роль играет формулировка задач и поиск эффективных алгоритмов и структур данных для их решения. Некоторые рассмотренные нами алгоритмы можно применить к ним непосредственно, но более важно то, что в них можно использовать общий подход — проектирование, реализацию и анализ алгоритмов, — который лежит в основе данной книги и многократно доказал свою практическую ценность. Этот принцип распространяется за пределы компьютерных наук на многие другие области исследований, от игр и музыки до лингвистики, финансов и исследования нервной деятельности.

Разработано уже очень много важных и полезных алгоритмов, поэтому становится важным изучение не только самих алгоритмов, ни взаимосвязей между ними. Мы закончим этот раздел (и книгу тоже) введением в *теорию алгоритмов*, с особым вниманием к *трудноразрешимости* и вопросу P = NP?, который до сих пор является ключевым при рассмотрении практических задач, которые жизнь ставит перед нами.

6.1. Событийное моделирование

Наш первый пример представляет собой фундаментальное научное приложение — моделирование поведения системы движущихся частиц в соответствии с законами упругого столкновения. Ученые используют такие системы для уяснения и предсказания свойств физических систем. Эта парадигма охватывает поведение молекул газа, динамику химических реакций, диффузию атомов, сферическую упаковку, устойчивость околопланетных колец, фазовые переходы некоторых элементов, одномерные системы с внутренней гравитацией, распространение фронтов и многие другие явления. Области применения лежат в диапазоне от молекулярной динамики, где объектами являются крошечные субатомные частицы, до астрофизики, где объекты — огромные небесные тела.

Для работы с этой задачей требуются некоторые университетские знания, немного проектирования ПО и немного алгоритмов. Большую часть физики мы выносим в упражнения в конце данного раздела, чтобы не отвлекаться от основной темы — использование фундаментальных алгоритмических средств (пирамидальных очередей с приоритетами) для прикладного моделирования, которые позволяют выполнять иначе невозможные вычисления.

Модель жестких дисков

Мы начнем с идеализированной модели движения атомов или молекул в контейнере со следующими основными качествами.

- Движущиеся *частицы* взаимодействуют друг с другом и *стенками* посредством упругих столкновений.
- Каждая частица представляет собой диск с известными позицией, скоростью, массой и радиусом.
- Никакие другие силы на частицы не влияют.

Эта простая модель играет главную роль в *статистической механике* — науке, которая связывает макроскопические наблюдаемые величины (такие как температура и давление) с микроскопической динамикой (движением отдельных атомов и молекул). Максвелл и Больцман использовали эту модель для вывода распределений скоростей взаимодействующих молекул в виде функции от температуры, а Эйнштейн применял ее для объяснения броуновского движения частиц пыльцы в воде. Из предположения об отсутствии влияния других сил следует, что между столкновениями частицы двигаются по прямолинейным траекториям с постоянной скоростью. Но в данную модель можно добавлять и другие силы. Например, добавив трение и вращение, можно более точно смоделировать движение таких знакомых физических объектов, как шары на бильярдном столе.

Временное моделирование

Наша основная цель — просто использовать модель, т.е. иметь возможность отслеживать позиции и скорости всех частиц в течение времени. Основное вычисление, которое необходимо при этом выполнять: для первоначальных позиций и скоростей в некото-

25 Зак. 3239



Рис. 6.1.1. Временное моделирование

Рис. 6.1.2. Фундаментальная проблема при временном моделировании

рый момент t изменить их так, чтобы они описывали ситуацию в последующий момент $t + dt$ для некоторого заданного промежутка dt (рис. 6.1.1). Если частицы расположены достаточно далеко друг от друга и от стенок и до момента $t + dt$ не произойдет ни одного столкновения, то вычисление выполняется легко: поскольку частицы движутся по прямолинейным траекториям, достаточно использовать их скорости, чтобы найти их позиции. Но учет столкновений существенно усложняет задачу (рис. 6.1.2). Один из способов ее решения — *временное моделирование* — использует фиксированное значение dt. При вычислении каждого обновления позиций необходимо проверить все пары частиц, определить, занимают ли они одно и то же место, и затем откатиться до момента первого такого столкновения. В этот момент можно правильно изменить скорости двух сталкивающихся частиц (с помощью приведенных ниже вычислений). При моделировании поведения системы из большого количества частиц этот подход сопряжен со значительными вычислительными затратами: если dt измеряется в секундах (а обычно это доли секунды), то для моделирования системы из N частиц на протяжении 1 секунды требуется время, пропорциональное N^2/dt. Такая стоимость непозволительна (даже больше, чем обычные квадратичные алгоритмы), т.к. в приложениях, которые интересны ученым, N очень велико, а dt очень мало. Проблема в том, что если взять dt слишком малым, то придется выполнять много вычислений, а если взять его слишком большим, то можно пропустить моменты столкновений.

Событийное моделирование

Мы воспользуемся другим способом, который принимает во внимание только моменты столкновений. В частности, нас всегда интересует момент *следующего* столкновения (поскольку до этого момента можно просто изменять позиции частиц на основе их скоростей). Значит, нужно использовать очередь *событий* с приоритетами, где событие — это некоторый момент времени в будущем, когда может произойти столкновение двух частиц или частицы со стенкой. Приоритетом, назначаемым каждому событию, является его время, поэтому при *извлечении наименьшего элемента* из такой очереди мы получим момент очередного столкновения.

Предсказание столкновений

Как можно идентифицировать потенциальные столкновения? Всю необходимую для этого информацию дают скорости частиц. Пусть, например, в момент времени t частица радиусом s находится в единичном квадрате в позиции (r_x, r_y) и движется со скоростью (v_x, v_y). Рассмотрим вертикальную стенку $x = 1$ при y от 0 до 1 (рис. 6.1.3).

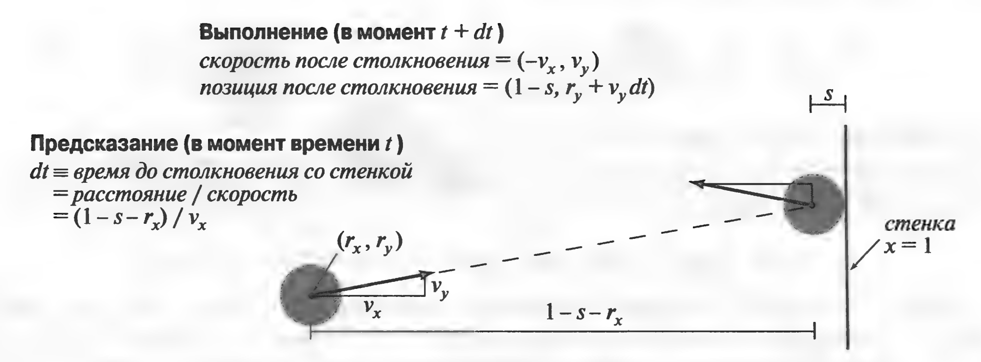
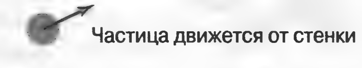


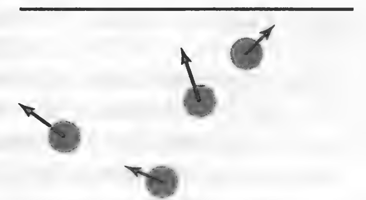
Рис. 6.1.3. Предсказание и выполнение столкновения частицы со стенкой

Нас интересует горизонтальная компонента движения, поэтому можно ограничиться рассмотрением x-компоненты позиции r_x и x-компоненты скорости v_x. Если значение v_x отрицательно, частица не столкнется со стенкой, но если оно положительно, столкновение возможно. Поделив горизонтальное расстояние до стены $(1 - s - r_s)$ на модуль горизонтальной компоненты скорости (v_x), мы получим, что эта частица столкнется со стенкой через $dt = (1 - s - r_x) / v_x$ единиц времени, когда она будет в позиции $(1 - s, r_y + v_y dt)$ — конечно, если до этого она не столкнется с другой частицей или горизонтальной стенкой. Поэтому мы помещаем в очередь с приоритетами значение $t + dt$, вместе с информацией, описывающей событие столкновения частицы со стенкой. Вычисления, предсказывающие моменты столкновения частицы с другими стенками, аналогичны (см. упражнение 6.1.1 и рис. 6.1.4). Вычисление для столкновения двух частиц также аналогично, хотя и сложнее. Часто бывает так, что вычисление предсказывает, что столкновение *не* произойдет (если частица движется от стенки, или если две частицы двигаются друг от друга, рис. 6.1.5) — тогда в очередь с приоритетами ничего заносить не надо. Для обработки другой типичной ситуации, когда предсказываемое столкновение находится в слишком далеком будущем, мы введем параметр `limit`, который определяет интересующий нас период времени, чтобы игнорировать все события, которые произойдут после него.



Рис. 6.1.4. Предсказуемые события







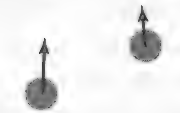


Рис. 6.1.5. Предсказуемые ситуации без столкновений

Выполнение столкновений

При столкновении его необходимо выполнить на основании физических формул, которые описывают поведение частиц после упругого столкновения с границей или другой частицей.

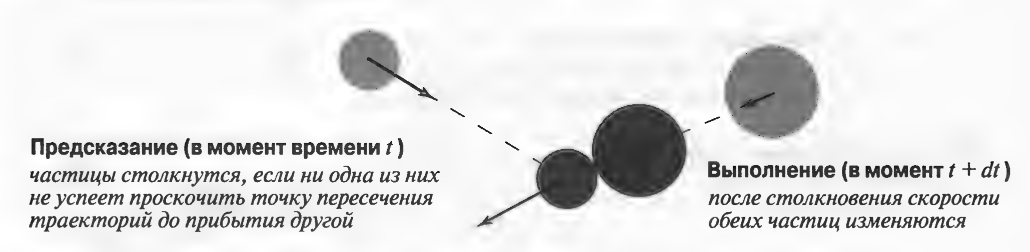


Рис. 6.1.6. Предсказание и выполнение столкновения двух частиц

В нашем примере, где частица сталкивается с вертикальной стенкой, в момент столкновения скорость частицы меняется с (v_x, v_y) на $(-v_x, v_y)$ (рис. 6.1.6). Формулы для остальных стенок аналогичны; похожи и формулы для столкновения двух частиц, но они несколько сложнее (см. упражнение 6.1.1).

Отмена событий



Рис. 6.1.7. Отмена события

Многие из предсказанных столкновений на самом деле не происходят, т.к. этому мешают другие столкновения (рис. 6.1.7). Для обработки этой ситуации мы будем использовать для каждой частицы переменную-счетчик, подсчитывающую количество столкновений, в которых она задействована. При выборке события из очереди с приоритетами для обработки мы проверяем, изменились ли счетчики, соответствующие этой частице, с момента создания события. Это так называемый *ленивый* способ отмены событий: если частица поучаствовала в некотором событии, мы оставляем связанные с ней, но уже неверные события в очереди с приоритетами и игнорируем их лишь при выборке. Другой — *энергичный* — способ состоит в удалении из очереди с приоритетами всех событий, в которых участвует любая частица после столкновения, и вычислении всех новых потенциальных столкновений для этой частицы. Для такого способа нужна более сложная очередь с приоритетами (с реализацией операции *удалить*).

Приведенное обсуждение подготовило почву для полного событийного моделирования движения частиц, взаимодействующих друг с другом согласно физическим законам упругого столкновения. В программной архитектуре нужно инкапсулировать реализацию в три класса: тип данных `Particle`, инкапсулирующий вычисления, связанные с частицами; тип данных `Event` для предсказанных событий; и клиент `CollisionSystem`, который выполняет непосредственно моделирование. Основой этого моделирования является очередь `MinPQ`, которая содержит упорядоченные по времени события. А теперь рассмотрим реализации `Particle`, `Event` и `CollisionSystem`.

Частицы

В упражнении 6.1.1 описаны принципы реализации типа данных для частиц на основе непосредственного применения законов движения Ньютона. Клиенту моделирования нужна возможность двигать частицы, рисовать их и выполнять ряд вычислений, связанных со столкновениями, как показано на рис. 6.1.8.

public class **Particle**		
	Particle()	*создание новой случайной частицы в единичном квадрате*
	Particle(double rx, double ry, double vx, double vy, double s, double mass)	*создание частицы с заданными параметрами: позиция, скорость, радиус и масса*
void	draw()	*отрисовка на чертеже*
void	move(double dt)	*изменение позиции через промежуток времени* dt
int	count()	*количество столкновений с участием данной частицы*
double	timeToHit(Particle b)	*время, через которое данная частица столкнется с частицей* b
double	timeToHitHorizontalWall()	*время, через которое данная частица столкнется с горизонтальной стенкой*
double	timeToHitVericaltWall()	*время, через которое данная частица столкнется с вертикальной стенкой*
void	bounceOff(Particle b)	*изменение скоростей частиц при столкновении*
void	bounceOffHorizontalWall()	*изменение скорости при ударе о горизонтальную стенку*
void	bounceOffVerticalWall()	*изменение скорости при ударе о вертикальную стенку*

Рис. 6.1.8. *API для объектов, описывающих движущиеся частицы*

Все три метода `timeToHit*()` возвращают значение `Double.POSITIVE_INFINITY` для (нередкого) случая, если столкновение не предвидится. Эти методы позволяют предсказать все будущие столкновения, связанные с заданной частицей, помещая в очередь с приоритетами соответствующие события, если они прогнозируются до заданного момента времени `limit`. При каждой обработке события, соответствующего столкновению двух частиц, мы вызываем метод `bounce()`, чтобы изменить скорости (обеих частиц) в результате столкновения, а при каждой обработке события, соответствующего столкновению частицы со стенкой, мы вызываем методы `bounceOff*()`.

События

Описание объектов, которые помещаются в очередь с приоритетами (события), мы инкапсулируем в приватном классе. Переменная экземпляров `time` содержит предсказанный момент события, а переменные экземпляров `a` и `b` содержат частицы, связанные с событием. У нас имеются три различных вида событий: частица может столкнуться с вертикальной стенкой, горизонтальной стенкой и другой частицей. Для разработки гладкого динамического отображения движения частиц мы добавили четвертый тип события — события перерисовки, которое представляет собой команду для отрисовки всех частиц в их текущих позициях. В реализации типа `Event` присутствует небольшая хитрость — значения частиц могут быть нулевыми, и этот факт используется для кодирования описанных четырех видов событий:

- ни a, ни b не равны null: столкновение двух частиц;
- a не равно null, a b равно null: столкновение a с вертикальной стенкой;
- a равно null, a b не равно null: столкновение b с горизонтальной стенкой;
- и a, и b равны null: событие перерисовки (вычерчивание всех частиц).

Это соглашение не очень-то вяжется с объектно-ориентированным программированием, но оно позволяет естественным образом упростить код клиента и приводит к реализации, показанной в листинге 6.1.1.

Листинг 6.1.1. Класс события для моделирования движения частиц

```
private class Event implements Comparable<Event>
{
   private final double time;
   private final Particle a, b;
   private final int countA, countB;
   public Event(double t, Particle a, Particle b)
   {//Создание нового события, которое должно произойти в момент t с объектами a и b
      this.time = t;
      this.a    = a;
      this.b    = b;
      if (a != null) countA = a.count(); else countA = -1;
      if (b != null) countB = b.count(); else countB = -1;
   }
   public int compareTo(Event that)
   {
      if      (this.time < that.time) return -1;
      else if (this.time > that.time) return +1;
      else return 0;
   }
   public boolean isValid()
   {
      if (a != null && a.count() != countA) return false;
      if (b != null && b.count() != countB) return false;
      return true;
   }
}
```

В реализации Event имеется еще одна хитрость: в переменных экземпляров countA и countB записывается количество столкновений, в которых задействована каждая частица, *на момент создания события*. Если эти счетчики не изменились к моменту извлечения события из очереди, можно выполнять моделирование события, но если за это время хотя бы один из счетчиков изменился, это значит, что событие следует отменить. Метод isValid() позволяет выполнить такую проверку в коде клиента.

Код моделирования

После инкапсуляции вычислительных подробностей в классах Particle и Event само моделирование требует уже совсем немного кода — см. класс CollisionSystem в листингах 6.1.3 и 6.1.4. Основной объем вычислений сконцентрирован в методе predictCollisions(), который приведен в листинге 6.1.2. Этот метод вычисляет все возможные будущие столкновения, в которых может участвовать частица a (либо с другой частицей, либо со стенкой), и заносит соответствующие события в очередь с приоритетами.

Листинг 6.1.2. Предсказание столкновений с другими частицами

```
private void predictCollisions(Particle a, double limit)
{
   if (a == null) return;
   for (int i = 0; i < particles.length; i++)
   {  // Занесение столкновений с particles[i] в pq.
      double dt = a.timeToHit(particles[i]);
      if (t + dt <= limit)
         pq.insert(new Event(t + dt, a, particles[i]));
   }
   double dtX = a.timeToHitVerticalWall();
   if (t + dtX <= limit)
      pq.insert(new Event(t + dtX, a, null));
   double dtY = a.timeToHitHorizontalWall();
   if (t + dtY <= limit)
      pq.insert(new Event(t + dtY, null, a));
}
```

Листинг 6.1.3. Событийное моделирование сталкивающихся частиц (вспомогательный класс)

```
public class CollisionSystem
{
   private class Event implements Comparable<Event>
   {  /* См. текст. */  }

   private MinPQ<Event> pq;          // очередь с приоритетами
   private double t  = 0.0;          // моделирование часов
   private Particle[] particles;     // массив частиц

   public CollisionSystem(Particle[] particles)
   {  this.particles  = particles;  }

   private void predictCollisions(Particle a, double limit)
   {  /* См. текст. */  }

   public void redraw(double limit, double Hz)
   { // Перерисовка события: перерисовка всех частиц.
     StdDraw.clear();
     for(int i = 0; i < particles.length; i++) particles[i].draw();
     StdDraw.show(20);
     if (t < limit)
        pq.insert(new Event(t + 1.0 / Hz, null, null));
   }
   public void simulate(double limit, double Hz)
   {  /* См. листинг 6.1.4. */  }
   public static void main(String[] args)
   {
      StdDraw.show(0);
      int N = Integer.parseInt(args[0]);
      Particle[] particles = new Particle[N];
      for (int i = 0; i < N; i++)
         particles[i] = new Particle();
      CollisionSystem system = new CollisionSystem(particles);
      system.simulate(10000, 0.5);
   }
}
```

Этот класс реализует очередь с приоритетами, которая моделирует поведение системы частиц в течение некоторого времени. Клиент тестирования `main()` принимает аргумент командной строки N, создает N случайных частиц, создает объект `CollisionSystem`, состоящий из созданных частиц, и вызывает метод моделирования `simulate()`. Переменными экземпляров являются очередь с приоритетами для нужд моделирования, время и частицы.

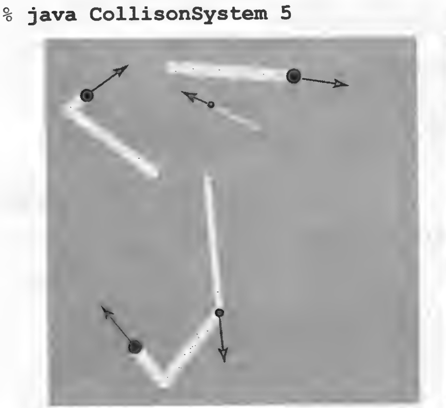
Листинг 6.1.4. Событийное моделирование сталкивающихся частиц (основной цикл)

```
public void simulate(double limit, double Hz)
{
  pq = new MinPQ<Event>();
  for (int i = 0; i < particles.length; i++)
    predictCollisions(particles[i], limit);
  pq.insert(new Event(0, null, null));      // Добавление перерисовки события.

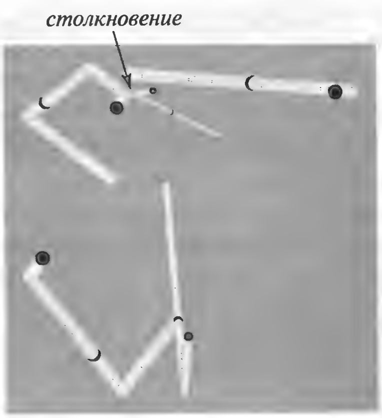
  while (!pq.isEmpty())
  {  // Обработка одного события для выполнения моделирования.
    Event event = pq.delMin();
    if (!event.isValid()) continue;
    for (int i = 0; i < particles.length; i++)
      particles[i].move(event.time - t);    // Обновление позиций частиц
    t = event.time;                         //   и времени.
    Particle a = event.a, b = event.b;
    if     (a != null && b != null) a.bounceOff(b);
    else if (a != null && b == null) a.bounceOffHorizontalWall();
    else if (a == null && b != null) b.bounceOffVerticalWall();
    else if (a == null && b == null) redraw(limit, Hz);
    predictCollisions(a, limit);
    predictCollisions(b, limit);
  }
}
```

Этот метод выполняет основное событийное моделирование. Вначале очередь с приоритетами заполняется событиями, представляющими все предсказанные будущие столкновения, в которых задействована каждая частица. Затем в основном цикле из очереди по одному выбираются события, изменяется время и позиции частиц, и добавляются новые события, которые отражают последние изменения.

```
% java CollisonSystem 5
```



столкновение



Основой моделирования является метод `simulate()`, приведенный в листинге 6.1.4. Вначале вызывается метод `predictCollisions()` для каждой частицы, чтобы заполнить очередь с приоритетами потенциальными столкновениями как с другими частицами, так и со стенками. Затем выполняется основной цикл событийного моделирования, который выполняет следующие шаги.

- Удаляется ближайшее событие (с минимальным приоритетом t).
- Если событие неверно, оно игнорируется.
- Все частицы двигаются до момента времени t по прямолинейным траекториям.
- Обновляются скорости частиц, которые участвовали в столкновении.
- Вызывается метод `predictCollisions()`, чтобы предсказать будущие столкновения только что столкнувшихся частиц и вставить в очередь соответствующие события.

Это моделирование может служить основой для вычисления всех видов свойств системы, которые только могут понадобиться — см. упражнения в конце раздела. Например, одним из фундаментальных свойств является величина давления, оказываемого частицами на стенки. Для вычисления этого давления можно отслеживать количество и модуль скорости при столкновениях со стенками (это легко сделать, если знать массы и скорости частиц) и подсчитать суммарную величину. Для подсчета температуры выполняется аналогичное вычисление.

Производительность

Как было сказано в начале данного раздела, нас интересует такая организация событийного моделирования, которая позволит избежать вычислительно сложного внутреннего цикла, присущего временн*о*му моделированию.

Утверждение А. Событийное моделирование N сталкивающихся частиц требует не более N^2 операций в очереди с приоритетами на этапе инициализации и не более N операций в очереди с приоритетами на каждое столкновение (с одной дополнительной операцией в очереди с приоритетами для каждого неверного столкновения).

Доказательство. Непосредственно следует из кода.

Наша стандартная реализация очереди с приоритетами с гарантированным логарифмическим временем обработки одной операции из раздела 2.4 позволяет получить линейно-логарифмическое время обработки одного столкновения. Значит, вполне возможно выполнять моделирования с большими количествами частиц.

Событийное моделирование применимо в бесчисленных других областях, где требуется физическое моделирование движущихся объектов, от молекулярной физики до астрофизики и робототехники. Такие приложения могут потребовать расширения модели: введения других видов физических тел, работы в трех измерениях, рассмотрения других сил и многих других модификаций. С каждым таким расширением связаны особые вычислительные сложности. Событийный подход позволяет выполнять более надежное, точное и эффективное моделирование, чем многие другие подходы, а эффективность пирамидальной очереди с приоритетами позволяет выполнять вычисления, невозможные при других способах.

Моделирование играет важную роль, помогая исследователям выяснить свойства естественного мира во всех областях науки и техники. Приложения могут находиться в диапазоне от производственных процессов до биологических систем, от финансовых систем до сложных технологических структур — все просто невозможно перечислить. Для значительной части этих приложений дополнительная эффективность, предоставляемая типом данных пирамидальной очереди с приоритетами или быстрым алгоритмом сортировки, может означать существенную разницу в качестве и охвате моделирования.

Упражнения

6.1.1. Завершите реализацию `predictCollisions()` и `Particle`, как описано в тексте. Имеются три уравнения, описывающие упругое столкновение двух твердых дисков: (*а*) сохранение линейного момента, (*б*) сохранение кинетической энергии и (*в*) при столкновении силы действуют в точке соприкосновения перпендикулярно поверхности (в отсутствие трения и вращения). Подробнее см. на сайте книги.

6.1.2. Разработайте вариант классов `CollisionSystem`, `Particle` и `Event` с обработкой столкновений нескольких частиц. Такие столкновения важны при моделировании игры в бильярд. (Упражнение сложное!)

6.1.3. Разработайте вариант классов `CollisionSystem`, `Particle` и `Event` для трех измерений.

6.1.4. Обдумайте следующую идею повышения производительности метода `simulate()` в классе `CollisionSystem`. Область действия делится на прямоугольные ячейки, и добавляется новый вид события, чтобы в любой момент времени предсказывать столкновения с частицами лишь из девяти соседних ячеек. Этот способ снижает количество вычисляемых предсказаний за счет слежения за перемещением частиц из ячейки в ячейку.

6.1.5. Введите в класс `CollisionSystem` концепцию *энтропии* и используйте ее для подтверждения классических результатов.

6.1.6. *Броуновское движение.* В 1827 г. ботаник Роберт Броун наблюдал в микроскоп движение частиц цветочной пыльцы, взвешенных в воде. Он заметил, что эти частицы находятся в беспорядочном движении, которое впоследствии назвали броуновским движением. Это явление обсуждалось, но математическое объяснение для него предложил лишь Эйнштейн в 1905 г.: движение частиц цветочной пыльцы обусловлено миллионами крошечных молекул, сталкивающихся с этими частицами. Проведите моделирование, подтверждающее это объяснение.

6.1.7. *Температура.* Добавьте в класс `Particle` метод `temperature()`, который возвращает произведение массы частицы на квадрат ее скорости, деленное на dk_B, где $d = 2$ — размерность пространства, а $k_B = 1{,}3806503 \cdot 10^{-23}$ — постоянная Больцмана. Температура системы равна среднему значению этих величин. Затем добавьте метод `temperature()` в класс `CollisionSystem` и напишите драйвер, который периодически выводит на график значение температуры, чтобы проверить ее неизменность.

6.1.8. *Распределение Максвелла-Больцмана.* Скорости частиц в модели твердых дисков подчиняется *распределению Максвелла-Больцмана* (при условии, что система термоизолирована, а частицы достаточно тяжелы, чтобы не учитывать квантовомеханические эффекты). В случае двух измерений это распределение называется распределением Релея. Вид графика распределения зависит от температуры. Напишите драйвер, который вычисляет гистограмму скоростей частиц и выводит ее для различных температур.

6.1.9. *Произвольная форма.* Молекулы двигаются очень быстро (быстрее реактивного самолета), однако диффундируют медленно, т.к. сталкиваются с другими молекулами, которые изменяют их движение. Добавьте в модель учет формы сосуда, когда два сосуда с первоначально различными частицами соединены трубкой. Выполните моделирование и определите долю частиц каждого типа в каждом сосуде как функцию времени.

6.1.10. *Возврат в начало.* После выполнения моделирования измените скорости всех частиц на обратные и снова запустите систему — она должна вернуться в исходное состояние. Определите ошибку округления, измерив разность между окончательным и исходным состояниями системы.

6.1.11. *Давление.* Добавьте в класс `Particle` метод `pressure()`, который измеряет давление, подсчитывая количество столкновений со стенками и компоненты скоростей частиц, перпендикулярные стенкам. Давление всей системы равно сумме этих величин. Затем добавьте метод `pressure()` в класс `CollisionSystem` и напишите клиент для проверки уравнения $pv = nRT$.

6.1.12. *Реализация индексной очереди с приоритетами.* Разработайте вариант класса `CollisionSystem`, где используется индексная очередь с приоритетами, которая гарантирует, что размер очереди будет не более чем линейно зависеть от количества частиц (вместо квадратичной зависимости в худшем случае).

6.1.13. *Производительность очереди с приоритетами.* Добавьте нужные средства и протестируйте работу метода `pressure()` при различных температурах, чтобы обнаружить узкое место в вычислениях. При необходимости попробуйте переключиться на другую реализацию очереди с приоритетами, чтобы повысить производительность при высоких температурах.

6.2. B-деревья

В главе 3 мы убедились, что алгоритмы, позволяющие обращаться к элементам в огромных коллекциях данных, имеют большую практическую важность. Поиск — это фундаментальная операция в наборах данных, а при большом объеме данных такой поиск занимает заметную часть ресурсов многих вычислительных сред. С появлением Интернета появилась возможность получать огромные объемы информации, которые могут как-то относиться к нашей задаче, и основная трудность состоит в эффективном поиске в этой информации. В данном разделе мы опишем еще одно расширение алгоритмов на сбалансированных деревьях из раздела 3.3, позволяющее поддерживать *внешний поиск* в таблицах имен, которые хранятся на диске или в веб-сети, и поэтому могут иметь объем, гораздо больший, чем рассмотренные нами раньше (которые должны были входить в оперативную память). Современные программные системы размывают различие между локальными файлами и веб-страницами, которые могут храниться на удаленном компьютере, поэтому количество данных, в которых может потребоваться поиск, практически не ограничено. Методы, которые мы сейчас рассмотрим, могут поддерживать операции поиска и вставки в таблицах имен, содержащих триллионы элементов и более, с помощью лишь четырех-пяти обращений к небольшим блокам данных.

Модель стоимости

Механизмы хранения данных сильно отличаются друг от друга и постоянно развиваются, поэтому мы возьмем для использования простую модель, которая содержит все важные для нас аспекты. Мы будем применять термин *страница* для непрерывного блока данных и термин *проба* для первого обращения к странице. Мы предполагаем, что для обращения к странице необходимо прочитать ее содержимое в локальную память, а последующие обращения к ней будут иметь гораздо меньшую стоимость. Страница может быть файлом на локальном компьютере, или веб-страницей на удаленном компьютере, или частью файла на сервере, или чем-то еще. И нам нужно разработать реализации поиска, которые используют небольшое количество проб для нахождения любого заданного ключа. Мы не будем делать конкретных предположений о размере страницы и об отношении времени, необходимого для пробы (которая, скорее всего, требует взаимодействия с удаленным устройством), к времени, необходимому для последующих обращений к элементам блока (что, скорее всего, происходит на локальном процессоре). В типичных ситуациях эти значения обычно имеют порядок 100, или 1000, или 10000 — а большая точность и не нужна, поскольку алгоритмы не очень чувствительны к этим значениям.

Модель стоимости для B-деревьев. При изучении алгоритмов для внешнего поиска мы подсчитываем *обращения к страницам* (количество чтений или записей этих страниц).

B-деревья

В основе нашего подхода лежит расширение структуры 2-3-деревьев, описанной в разделе 3.3, но с одной существенной разницей: вместо хранения данных дерево будет содержать *копии* ключей и связанные с ними ссылки. Этот подход позволяет легче от-

делить индекс от самой таблицы — примерно как индекс книги. Как и в случае 2-3-деревьев, мы накладываем верхние и нижние границы на количество пар ключ-ссылка, которые могут храниться в каждом узле. Для этого мы выберем параметр M (по соглашению, четный) и будем строить многочастные деревья, в которых каждый узел должен содержать *не более* $M-1$ пар ключ-ссылка (M должно быть не очень велико, чтобы M-частный узел помещался в странице) и *не менее* $M/2$ пар ключ-ссылка (чтобы обеспечить ветвление, достаточное для поддержания коротких путей поиска). Возможным исключением из этого правила является корень, который может иметь меньше $M/2$ пар ключ-ссылка, но не менее 2. Такие деревья называются *В-деревьями* по имени Байера (Bayer) и Мак-Крейта (McCreight), которые в 1970 г. впервые исследовали применение многочастных сбалансированных деревьев для внешнего поиска. Некоторые относят термин *В-дерево* только к той структуре, которую строит алгоритм, предложенный Байером и Мак-Крейтом, но мы будем использовать его в качестве обобщенного термина для структур данных, основанных на многочастных сбалансированных деревьях поиска с фиксированным размером страницы. Значение M мы будем задавать с помощью термина *В-дерево порядка* M. В В-дереве порядка 4 каждый узел содержит не более 3 и не менее 2 пар ключ-ссылка; в В-дереве порядка 6 каждый узел содержит не более 5 и не менее 3 пар ключ-ссылка (кроме, возможно, корня, который может содержать 2 пары) и т.д. Причины особых условий для корня при больших M станут понятны, когда мы будем подробно рассматривать алгоритм построения дерева.

Соглашения

Для демонстрации базовых механизмов мы рассмотрим реализацию (упорядоченного) множества `SET` (с ключами, но без значений). Получение на ее основе упорядоченной таблицы поиска, которая связывает ключи со значениями, является полезным упражнением (см. упражнение 6.2.16). Нам нужно поддерживать методы `add()` и `contains()` для потенциально огромных множеств ключей. Упорядочение ключей необходимо потому, что нам нужно получить обобщение деревьев поиска, которые основаны на упорядоченных ключах. Полезным упражнением также будет расширение нашей реализации для поддержки других упорядоченных операций. В приложениях внешнего поиска индекс обычно хранится отдельно от данных. Для В-деревьев мы будем делать так с помощью двух различных видов узлов (рис. 6.2.1):

- *внутренние* узлы, которые связывают копии ключей со страницами;
- *внешние* узлы, которые содержат ссылки на сами данные.

Каждый ключ во внутреннем узле связан с другим узлом — корнем дерева, содержащего все ключи, которые *больше или равны* этому ключу и *меньше* следующего большего ключа, если такой есть. В В-деревьях удобно использовать специальный ключ, называемый *сигнальным*: он меньше всех других ключей, первоначально помещается в корневой узел и связан с деревом, содержащим все ключи. Таблица имен не содержит одинаковые ключи, но мы используем копии ключей (во внутренних узлах) для управления поиском. (В наших примерах содержатся однобуквенные ключи и звездочка `*`, обозначающая сигнальный ключ, который меньше всех остальных ключей.) Эти соглашения несколько упрощают код и, таким образом, представляют удобную (и широко используемую) альтернативу перемешиванию всех данных со ссылками во внутренних узлах, как это делалось в других деревьях поиска.

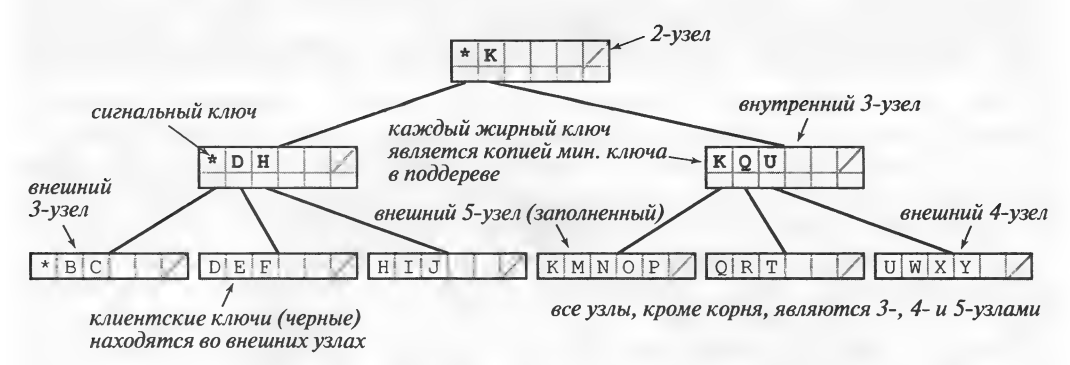


Рис. 6.2.1. Структура В-дерева (M = 6)

Поиск и вставка

Поиск в В-дереве основан на рекурсивном поиске в уникальном поддереве, в котором может находиться искомый ключ. Каждый поиск завершается во внешнем узле, который содержит ключ в том и только том случае, когда он присутствует в наборе (рис. 6.2.2). Можно завершить поиск, сигнализировав *попадание*, если копия искомого ключа встретится во внутреннем узле, но мы будем всегда проводить поиск до внешнего узла: это упрощает расширение кода до реализации упорядоченной таблицы имен (кроме того, этот событие нечасто происходит при большом значении *M*).

Для конкретности рассмотрим поиск в В-дереве порядка 6. Оно состоит из 3-узлов с 3 парами ключ-ссылка, 4-узлов с 4 парами ключ-ссылка и 5-узлов с 5 парами ключ-ссылка (корень может быть 2-узлом). Выполнение поиска мы начинаем с корня и переходим из одного узла в другой с помощью соответствующих ссылок. В конце концов, процесс поиска приведет нас в страницу, содержащую ключи на нижнем уровне дерева. Поиск завершается попаданием, если ключ присутствует на этой странице, и промахом, если его там нет.

Как и в случае 2-3-деревьев, можно использовать рекурсивный код для вставки нового ключа в нижнюю часть дерева (рис. 6.2.3). Если места для ключа нет, мы разрешаем временное переполнение узла на нижнем уровне (он становится 6-узлом), а затем выполняем разбиение 6-узлов, поднимаясь по дереву после завершения рекурсивных вызовов. Если корень является 6-узлом, мы преобразуем его в 3-узел, соединенный с двумя 3-узлами; во всех других местах дерева любой k-узел, соединенный с 6-узлом, заменяется $(k + 1)$-узлом, соединенным с двумя 3-узлами. Замена 3 на $M/2$ и 6 на M в этом описании преобразует его в описание поиска и вставки в В-деревьях порядка M:

Определение. *В-дерево порядка M* (где M — четное положительное целое число) представляет собой дерево, которое либо является внешним k-узлом (с k ключами и связанной с ними информацией), либо состоит из внутренних k-узлов (каждый, k ключами и k ссылками на В-деревья, представляет k интервалов, разграниченных ключами) со следующими структурными свойствами: каждый путь от корня до внешнего узла должен иметь одну и ту же длину (идеальный баланс); и k должно быть от 2 до $M-1$ в корне и от $M/2$ до $M-1$ в любом другом узле.

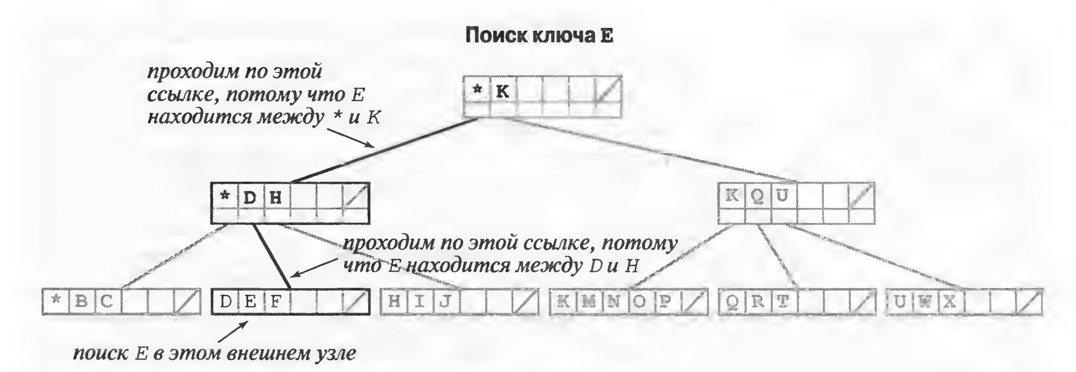


Рис. 6.2.2. Поиск в В-дереве (M = 6)

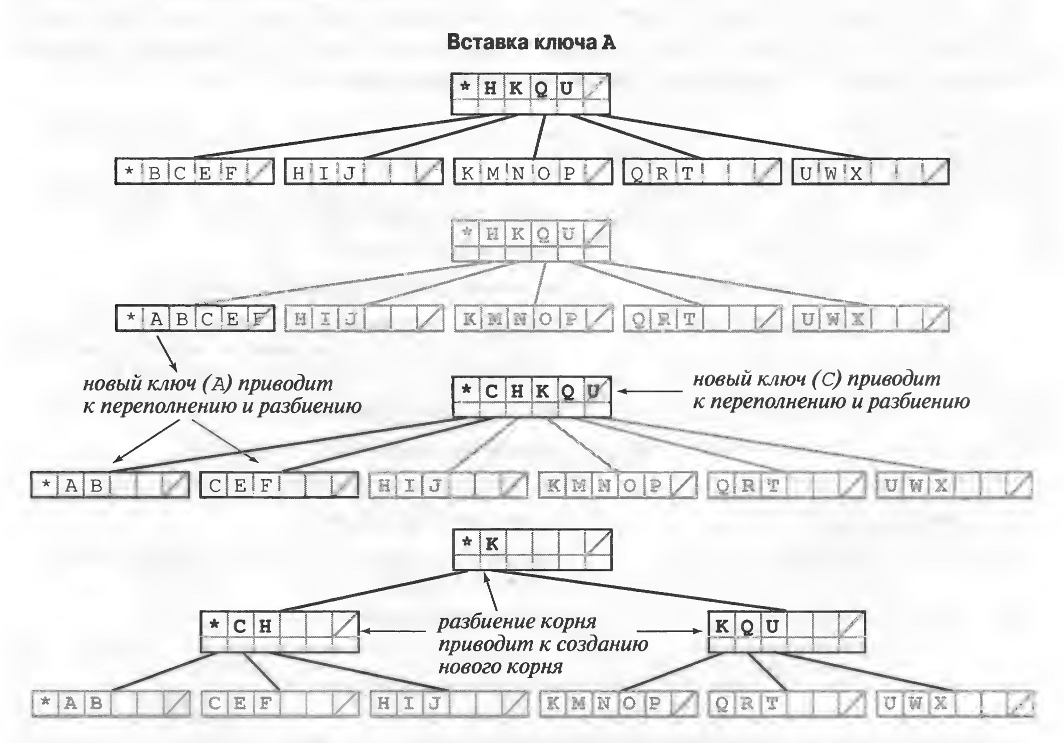


Рис. 6.2.3. Вставка нового ключа в В-дерево

Представление

Как только что было сказано, у нас имеется большая свобода в выборе конкретных представлений для узлов В-деревьев. Мы инкапсулируем результаты этого выбора в API Page (рис. 6.2.4), который связывает ключи со ссылками на объекты Page и поддерживает операции, необходимые для проверки на переполнение страниц, их разбиения и различения внутренних и внешних узлов. Объект Page можно рассматривать как таблицу имен, хранимую на внешнем носителе (в файле на компьютере или в сети).

Термины *открытие* и *закрытие* в этом API соответствуют процессу копирования внешней страницы во внутреннюю память и записи ее содержимого назад (при необходимости). Метод `put()` для внутренних страниц представляет собой операцию с таблицей имен, которая связывает заданную страницу с минимальным ключом в дереве с корнем в этой странице. Методы `put()` и `contains()` для внешних страниц похожи на соответствующие операции класса `SET`. Основой любой реализации является метод `split()`, который разбивает заполненную страницу, пересылая $M/2$ пар ключ-значение с рангом больше $M/2$ в новый объект `Page`, и возвращает ссылку на эту страницу. В упражнении 6.2.2 рассматривается реализация `Page` на основе `BinarySearchST`, которая строит B-деревья в памяти, как и наши другие реализации поиска. В некоторых системах это вполне может сойти за реализацию внешнего поиска, т.к. система виртуальной памяти может позаботиться обо всех обращениях к диску. А более типичные практические реализации могут содержать код чтения и записи страниц, зависящий от оборудования. Упражнение 6.2.6 приглашает подумать над реализацией класса `Page` с помощью веб-страниц. В тексте мы не будем отвлекаться на такие детали, чтобы обращать больше внимания на пользу концепции B-деревьев в широком диапазоне настроек.

public class	**Page**<Key>	
	`Page(boolean bottom)`	*создание и открытие страницы*
`void`	`close()`	*закрытие страницы*
`void`	`add(Key key)`	*занесение ключа в (внешнюю) страницу*
`void`	`add(Page p)`	*открытие страницы* `p` *и помещение элемента в эту (внутреннюю) страницу, которая связывает наименьший ключ в* `p` *с самой страницей* `p`
`boolean`	`isExternal()`	*является ли страница внешней?*
`boolean`	`contains(Key key)`	*находится ли ключ* `key` *в странице?*
`Page`	`next(Key key)`	*поддерево, которое может содержать ключ*
`boolean`	`isFull()`	*переполнена ли страница?*
`Page`	`split()`	*перемещение половины ключей с большим рангом в новую страницу*
`Iterable<Key>`	`keys()`	*итератор для ключей в странице*

Рис. 6.2.4. API для страницы B-дерева

После такой подготовки код класса `BTreeSET`, приведенный в листинге 6.2.1, выглядит весьма просто. Метод `contains()` реализован рекурсивно, он принимает в качестве аргумента объект `Page` и обрабатывает три случая.

- Если страница внешняя, и ключ находится в странице, возвращается `true`.
- Если страница внешняя, и ключ не находится в странице, возвращается `false`.
- Иначе выполняется рекурсивный вызов для поддерева, которое может содержать ключ.

Листинг 6.2.1. Алгоритм 6.1. Реализация В-дерева

```
public class  BTreeSET<Key extends Comparable<Key>>
{
  private Page root = new Page(true);

  public BTreeSET(Key sentinel)
  {  put(sentinel);  }

  public boolean contains(Key key)
  {  return contains(root, key);  }

  private boolean contains(Page h, Key key)
  {
    if (h.isExternal()) return h.contains(key);
    return contains(h.next(key), key);
  }

  public void add(Key key)
  {
    put (root, key);
    if (root.isFull())
    {
      Page lefthalf = root;
      Page righthalf = root.split();
      root = new Page(false);
      root.put(lefthalf);
      root.put(righthalf);
    }
  }

  public void add(Page h, Key key)
  {
    if (h.isExternal()) {  h.put(key); return;  }
    Page next = h.next(key);
    put(next, key);
    if (next.isFull())
      h.put(next.split());
    next.close();
  }
}
```

Этот код реализует многочастное сбалансированное В-дерево поиска, описанное в тексте, на основе типа данных `Page`. Данный тип поддерживает поиск, связывая ключ с поддеревьями, которые могут содержать этот ключ, и поддерживает вставку, включая проверку на переполнение страницы и метод для ее разбиения.

Для метода `put()` используется та же рекурсивная структура, но в ней выполняется вставка ключа на нижний уровень, если он не найден при поиске, а потом разбиения всех заполненных узлов при подъеме по дереву.

26 Зак. 3239

Производительность

Наиболее важным свойством В-деревьев является то, что для разумных значений параметра M стоимость поиска *постоянна* для всех практических целей.

Утверждение Б. Поиск или вставка в В-дереве порядка M с N элементами требует от $\log_M N$ до $\log_{M/2} N$ проб — для практических целей это постоянное количество.

Доказательство. Это свойство следует из того, что все узлы внутри дерева (т.е. не корень и не внешние узлы) содержат от $M/2$ до $M-1$ ссылок, поскольку они формируются разбиением полного узла с M ключами, и могут только расти в размере (когда разбивается дочерний узел). В лучшем случае эти узлы формируют полное дерево с коэффициентом ветвления $M-1$, откуда непосредственно следует заявленная граница. В худшем случае имеется корень с двумя элементами, каждый из которых указывает на полное дерево степени $M/2$. Логарифмирование по основанию M дает очень небольшое число: например, при $M = 1000$ высота дерева меньше 4, если N меньше 62.5 миллиарда.

В обычных ситуациях стоимость можно уменьшить на одну пробу, храня корень во внутренней памяти. При поиске на диске или в веб-сети этот шаг можно выполнять явно, прежде чем приступить к большому количеству поисков. В виртуальной памяти с кешированием корневой узел с большой вероятностью будет находиться в быстрой памяти, т.к. обращения к нему происходят чаще всего.

Память

В практических приложениях представляет интерес и объем памяти, занимаемой В-деревьями. По построению страницы пусты не более чем наполовину, поэтому в худшем случае В-деревья используют примерно в два раза больше памяти, чем абсолютно необходимо для данного набора ключей, плюс дополнительная память для ссылок. Для случайных ключей А. Яо (A. Yao) доказал в 1979 г. (с помощью математического анализа, который выходит за рамки данной книги), что среднее количество ключей в узле примерно равно $M \ln 2$, т.е. не используется приблизительно 44% памяти. Как и для многих других алгоритмов поиска, эта случайная модель неплохо предсказывает результаты для распределений ключей, которые обычно встречаются на практике.

Следствия из утверждения Б значительны и даже удивительны. Могли ли вы предположить, что возможна реализация поиска, которая может гарантировать стоимость в четыре-пять проб для поиска и вставки в файлы такого размера, что их хватит для всех разумных случаев? В-деревья широко используются потому, что они позволяют достичь такого практически идеального результата. В реальности основной трудностью при разработке реализаций является проверка наличия памяти для узлов В-дерева, но даже эта трудность становится все менее трудной по мере увеличения объема типичных запоминающих устройств.

Сразу приходят на ум многие разновидности базовой абстракции В-деревьев. Один класс таких разновидностей экономит время, упаковывая во внутренние узлы как можно больше ссылок на страницы — т.е. увеличивая коэффициент ветвления и уменьшая высоту дерева. Другой класс повышает эффективность использования памяти, по возможности комбинируя узлы с их соседями вместо разбиения.

Конкретный выбор разновидности и параметров алгоритма нужно выполнять с учетом конкретных устройств и приложений. Правда, мы ограничены лишь небольшой возможностью уменьшения количества проб, но даже такое уменьшение может оказаться очень важным в приложениях, где таблицы имеют огромные размеры и/или выполняется огромное количество запросов — т.е. в приложениях, для которых так эффективны В-деревья (рис. 6.2.5).

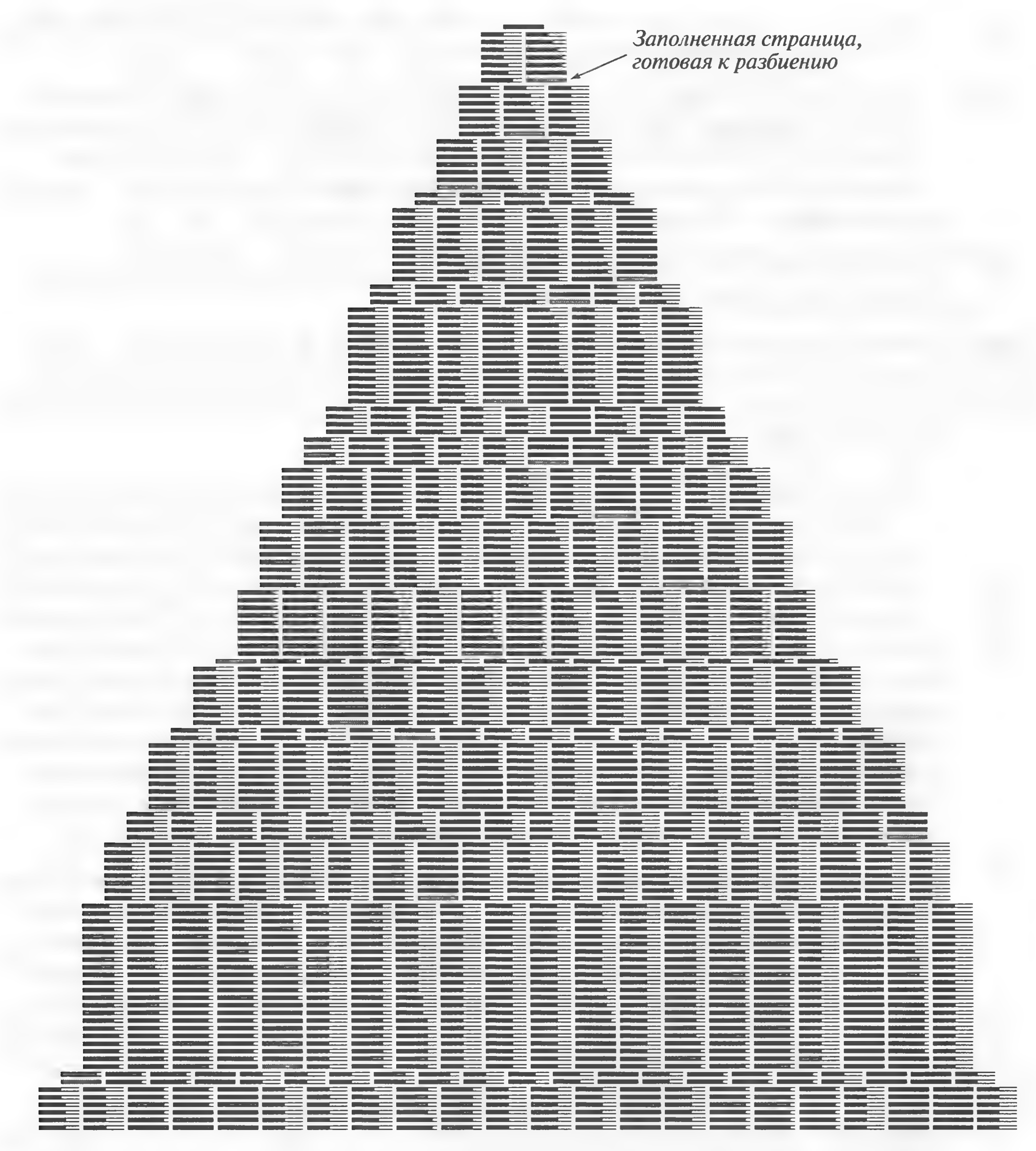


Рис. 6.2.5. Построение большого В-дерева

Упражнения

6.2.1. Пусть в случае дерева с тремя уровнями мы можем хранить ссылки во внутренней памяти: от b до $2b$ ссылок в страницах, представляющих внутренние узлы, и от c до $2c$ элементов в страницах, представляющих внешние узлы. Каково максимальное количество элементов, которое может содержаться в таком дереве, в виде функции от b и c?

6.2.2. Разработайте реализацию класса `Page`, в которой каждый узел B-дерева представлен в виде объекта `BinarySearchST`.

6.2.3. Разработайте на основе `BTreeSET` реализацию `BTreeST`, которая связывает ключи со значениями и поддерживает наш API для полностью упорядоченной таблицы имен с методами `min()`, `max()`, `floor()`, `ceiling()`, `deleteMin()`, `deleteMax()`, `select()`, `rank()` и разновидностями `size()` и `get()` с двумя аргументами.

6.2.4. Напишите программу, которая использует класс `StdDraw` для визуализации разрастания B-деревьев, как это сделано в тексте.

6.2.5. Оцените среднее количество проб на один поиск в B-дереве для S случайных поисков в типичной системе кеширования, где в памяти хранятся T страниц, к которым выполнялись последние поиски (и которые не увеличивают счетчик проб). Считайте, что S значительно больше T.

6.2.6. *Веб-поиск.* Разработайте реализацию класса `Page`, которая представляет узлы B-дерева в виде текстовых файлов на веб-страницах, для целей индексации веб-сети. Используйте файл с терминами поиска и вводите индексируемые страницы из стандартного ввода. Для управления процессом принимайте из командной строки параметр m и установите верхний предел в 10^m внутренних узлов (согласуйте это значение со своим системным администратором, прежде чем браться за большие значения m). Для именования внутренних узлов используйте m-значные числа. Например, при $m = 4$ имена узлов могут быть `BTreeNode0000`, `BTreeNode0001`, `BTreeNode0002` и т.д. Храните в страницах пары строк. Добавьте в API операцию `close()` для целей сортировки и вывода. Чтобы проверить работу вашей реализации, попробуйте найти информацию о себе и своих друзьях на веб-сайте вашего университета.

6.2.7. *B*-деревья.* Обдумайте следующую эвристику разбиения (*B*-дерево*) для B-деревьев. Когда нужно разбить узел, т.к. он содержит M элементов, мы объединяем этот узел с его соседом. Если этот соседний узел содержит k элементов, где $k < M-1$, то элементы перераспределяются по примерно $(M + k) / 2$ элементов в каждый узел. Иначе создается новый узел, и каждый из трех узлов будет содержать примерно $2M/3$ элементов. Кроме того, корень может разрастаться, чтобы вмещать до примерно $4M/3$ элементов, а при достижении этой границы он разбивается с созданием нового корня с двумя элементами. Приведите максимальные количества проб при выполнении поиска или вставки в B*-дерево порядка M с N элементами. Сравните эти границы с соответствующими значениями для B-деревьев (см. утверждение Б). Разработайте реализацию операции *вставить* для B*-деревьев.

6.2.8. Напишите программу для вычисления среднего количества внешних страниц в В-дереве порядка M, построенном N случайными вставками в первоначально пустое дерево. Выполните эту программу для разумных значений M и N.

6.2.9. Если ваша система поддерживает виртуальную память, обдумайте и выполните эксперименты, чтобы сравнить производительность В-деревьев с производительностью бинарного поиска, для случайных поисков в таблице имен очень большого размера.

6.2.10. Для вашей реализации класса `Page` во внутренней памяти из упражнения 6.2.2 экспериментально определите значение M, которое приводит к наиболее быстрому поиску в реализации В-дерева, поддерживающей случайные операции поиска в огромной таблице имен. Рассматривайте только значения M, кратные 100.

6.2.11. Экспериментально сравните длительность поисков для внутренних В-деревьев (используя значение M, определенное в предыдущем упражнении), хеширования с линейным опробованием и красно-черных деревьев, выполняя случайные операции поиска в огромной таблице имен.

6.3. Суффиксные массивы

Эффективные алгоритмы для обработки строк играют критическую роль в коммерческих приложениях и в научных вычислениях. От бесчисленных строк, определяющих веб-страницы, в которых ищут нужную информацию миллиарды пользователей, до обширных баз генетических данных, которые изучают ученые, чтобы раскрыть секрет жизни, вычислительные приложения XXI века все более ориентируются на обработку строк. Как обычно, с подобными задачами эффективно справляются некоторые классические алгоритмы, но разработаны также интересные новые алгоритмы. Сейчас мы опишем структуру данных и API, которые поддерживают некоторые такие алгоритмы. Но вначале мы обрисуем типичную (даже классическую) задачу из области обработки строк.

Максимальная повторяющаяся подстрока

Какова самая длинная подстрока, которая встречается хотя бы дважды в заданной строке? Например, в строке "`to be or not to be`" максимальная повторяющаяся подстрока — "`to be`". Подумайте немного, как бы вы могли решить такую задачу. Смогли бы вы найти такую подстроку в строке из миллионов символов?

Эта задача имеет простую формулировку и множество важных применений в сжатии данных, криптографии и компьютерном анализе музыкальных произведений. Например, при разработке больших программных систем часто используется стандартная техника — *рефакторинг кода*. Программисты зачастую составляют свои программы, копируя в них код из старых программ. Если большая программа разрабатывается длительное время, то замена такого повторяющегося кода на вызовы одной отдельной функции может заметно упростить понимание и сопровождение этой программы. Это усовершенствование можно выполнить, отыскивая в программе длинные повторяющиеся подстроки.

Еще одно применение можно обнаружить в вычислительной биологии. Можно ли найти идентичные фрагменты в заданном геноме? Здесь также требуется решать базовую вычислительную задачу — поиск в строке самой длинной повторяющейся подстроки. Обычно ученых интересуют более детальные вопросы (хотя природа повторяющихся подстрок — как раз то, в чем пытаются разобраться ученые), но, конечно, на такие вопросы ответить не легче, чем решить базовую задачу поиска максимальной повторяющейся подстроки.

Примитивное решение

В качестве разминки рассмотрим следующую простую задачу: даны две строки, и нужно найти их самый длинный общий *префикс* — т.е. самую длинную подстроку, которая находится в начале обеих строк. Например, самым длинным общим префиксом строк `acctgttaac` и `accgttaa` является `acc`. Код в листинге 6.3.1 представляет собой полезную отправную точку для рассмотрения более сложных задач: он отрабатывает за время, пропорциональное длине совпадающих подстрок.

Листинг 6.3.1. Максимальный общий префикс двух строк

```
private static int lcp(String s, String t)
{
   int N = Math.min(s.length(), t.length());
   for (int i = 0; i < N; i++)
      if (s.charAt(i) != t.charAt(i)) return i;
   return N;
}
```

Ну и как же теперь найти максимальную повторяющуюся подстроку в заданной строке? На ум сразу приходит примитивное решение на основе метода `lcp()`: нужно сравнить подстроку, начинающуюся в каждой позиции строки `i`, с подстрокой, начинающейся в каждой другой позиции `j`, и отслеживать максимальный размер совпадающих подстрок. Но для длинных строк этот код непригоден, потому что время его выполнения как минимум *квадратично* относительно длины строки: как обычно, количество различных пар `i` и `j` равно $N(N-1)/2$, поэтому количество вызовов `lcp()` при таком подходе будет равно $\sim N^2/2$. Использование этого решения для геномной последовательности из миллионов символов потребует триллионов вызовов `lcp()`, что просто невозможно.

Решение с сортировкой суффиксов

Сейчас мы рассмотрим хитроумный подход, который (неожиданным образом) использует сортировку для эффективного поиска максимальной повторяющейся подстроки, даже в случае строк огромного размера. Сначала с помощью Java-метода `substring()` создается массив строк, состоящих из суффиксов строки `s` (строк, которые начинаются в каждой позиции исходной строки и заканчиваются в ее конце), а затем этот массив сортируется. Алгоритм основан на том факте, что каждая подстрока является префиксом одного из суффиксов в этом массиве (рис. 6.3.1). После выполнения сортировки самые длинные (и не только) повторяющиеся подстроки находятся в соседних позициях массива. Теперь можно выполнить один проход по упорядоченному массиву, отслеживая самые длинные совпадающие префиксы в смежных строках. Этот подход гораздо эффективнее примитивного алгоритма, но прежде чем реализовать и анализировать его, мы рассмотрим еще одно применение сортировки суффиксов.

Входная строка

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
a a c a a g t t t a c  a  a  g  c
```

Суффиксы

```
 0 aacaagtttacaagc
 1 acaagtttacaagc
 2 caagtttacaagc
 3 aagtttacaagc
 4 agtttacaagc
 5 gtttacaagc
 6 tttacaagc
 7 ttacaagc
 8 tacaagc
 9 acaagc
10 caagc
11 aagc
12 agc
13 gc
14 c
```

Отсортированные суффиксы

 0 aacaagtttacaagc
11 aagc
 3 aagtttacaagc
 9 **acaag**c
 1 **acaag**tttacaagc
12 agc
 4 agtttacaagc
14 c
10 caagc
 2 caagtttacaagc
13 gc
 5 gtttacaagc
 8 tacaagc
 7 ttacaagc
 6 tttacaagc

Максимальная повторяющаяся подстрока

1 9
a**acaag**ttt**acaag**c

Рис. 6.3.1. Вычисление максимальной повторяющейся подстроки с помощью сортировки суффиксов

Индексация строки

Если нужно найти конкретную подстроку в большом тесте — например, при работе с текстовым редактором или на странице, просматриваемой в браузере — то выполняется *поиск подстроки*; эта задача рассматривалась в разделе 5.3. В рамках данной задачи мы считаем, что текст имеет относительно большой размер, и будем уделять основное внимание обработке *подстрок*, особенно возможности эффективного поиска этой подстроки в любом заданном тексте.

При вводе ключей поиска в строке веб-браузера выполняется *поиск по строковым ключам* — тема раздела 5.2. Поисковый механизм должен заранее создать индекс, т.к. он не может просматривать все веб-страницы в поисках ваших ключей. Как было сказано в разделе 3.5 (см. класс `FileIndex` в листинге 3.5.6), в идеале это должен быть инвертированный индекс (рис. 6.3.2 и 6.3.3), который связывает каждую возможную строку поиска со всеми содержащими ее веб-страницами — т.е. таблица имен, где каждая строка представляет собой строковый ключ, а каждое значение является набором указателей. Каждый такой указатель дает информацию, необходимую для нахождения вхождения ключа в веб-сети — URL веб-страницы и целочисленное смещение от начала этой страницы.

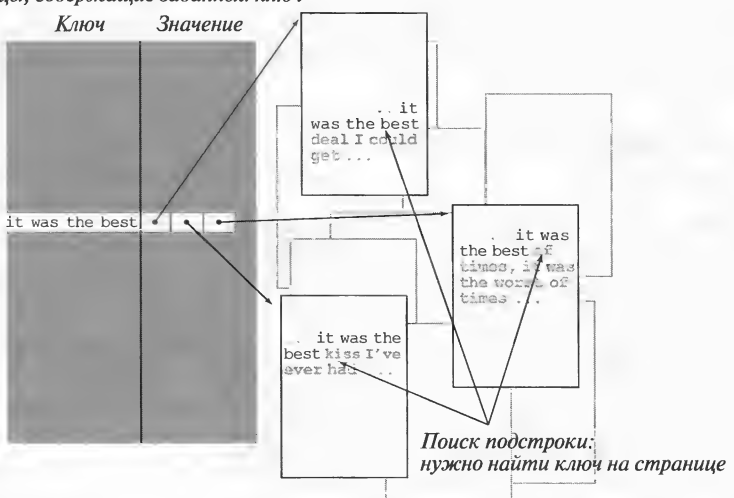


***Рис. 6.3.2.** Идеализированное представление типичного веб-поиска*

В реальности такая таблица имен была бы слишком большой, поэтому поисковые механизмы используют различные сложные алгоритмы для уменьшения ее размера. Один из способов состоит в ранжировании веб-страниц по важности (возможно, с использованием алгоритма наподобие PageRank, который был рассмотрен в разделе 3.5), после чего поиск выполняется только в страницах с высоким рангом, а не во всех страницах вообще. Еще один способ ограничить размер таблицы имен, в которой выполняется поиск строковых ключей, состоит в связывании URL-адресов со *словами* (строки, разграниченные пробельными символами), которые представляют собой ключи в заранее вычисленном индексе. Тогда при поиске какого-то слова поисковый механизм может применять индекс для нахождения (важных) страниц, содержащих ключи поиска (слова), а затем использовать поиск подстрок, чтобы найти это слово на каждой странице. Но если при

таком способе текст содержит слово "победа", то вы не найдете в нем слово "беда".

В некоторых приложениях *имеет* смысл строить индекс, который позволит найти *любую* строку в заданном тексте. Это может понадобиться при лингвистическом анализе важного литературного произведения, при изучении геномных последовательностей, которыми сейчас занимаются многие ученые, или просто для веб-страницы, к которой выполняется много обращений. В очередном идеальном случае индекс должен связывать все возможные подстроки из строки текста с каждой позицией, где встречаются эти подстроки (рис. 6.3.3). Основная проблема с таким идеальным случаем в том, что количество всех возможных подстрок слишком велико, чтобы разместить в таблице имен элементы для каждой из них: N-символьный текст содержит $N(N-1)/2$ подстрок. Таблица для примера на рис. 6.3.3 должна иметь элементы для подстрок `b`, `be`, `bes`, `best`, `best o`, `best of`, `e`, `es`, `est`, `est o`, `est of`, `s`, `st`, `st o`, `st of`, `t`, `t o`, `t of`, `o`, `of` и многих-многих других. Здесь также можно использовать сортировку суффиксов для решения этой задачи, как в нашей первой реализации таблицы имен с помощью бинарного поиска из раздела 3.1. Мы считаем каждый из N суффиксов ключом, создаем и сортируем массив ключей (суффиксов) и выполняем бинарный поиск в этом массиве, сравнивая каждый ключ с каждым суффиксом (рис. 6.3.4).

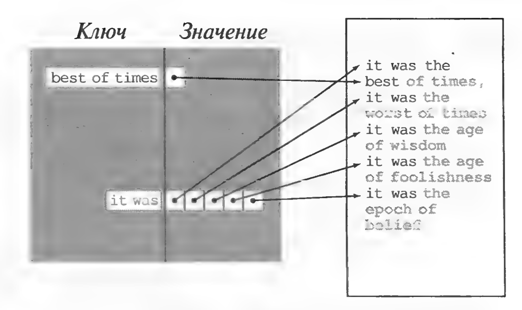


***Рис. 6.3.3.** Идеализированное представление индекса текстовой строки*

Суффиксы

0	it was the best of times it was the
1	t was the best of times it was the
2	was the best of times it was the
3	was the best of times it was the
4	**as the best of times it was the**
5	s the best of times it was the
6	the best of times it was the
7	the best of times it was the
8	he best of times it was the
9	e best of times it was the
10	best of times it was the
11	best of times it was the
12	est of times it was the
13	st of times it was the
14	t of times it was the
15	of times it was the
16	of times it was the
17	f times it was the
18	times it was the
19	times it was the
20	imes it was the
21	mes it was the
22	es it was the
23	s it was the
24	it was the
25	it was the
26	t was the
27	was the
28	was the
29	as the
30	s the
31	the
32	the
33	he
34	e

Отсортированный массив суффиксов

i	index(i)	lcp(i)	
0	10	0	best of times it was the
1	24	1	it was the
2	15	1	of times it was the
3	31	1	the
4	6	4	the best of times it was the
5	18	2	times it was the
6	27	1	was the
7	2	8	was the best of times it was the
8	29	0	as the
9	**4**	6	**as the best of times it was the**
10	11	0	best of times it was the
11	34	0	e
12	9	1	e best of times it was the
13	22	1	es it was the
14	12	2	est of times it was the
15	17	0	f times it was the
16	33	0	he
17	8	2	he best of times it was the
18	20	0	imes it was the
19	25	1	**it was the**
20	0	**10**	**it was the** best of times it was the
21	[illegible]	0	mes it was the
22	16	0	of times it was the
23	23	0	s it was the
24	30	2	s the
25	5	5	s the best of times it was the
26	13	1	st of times it was the
27	14	0	t of times it was the
28	26	1	t was the
29	1	9	t was the best of times it was the
30	32	1	**th**e
31	7	3	**th**e best of times it was the
32	19	1	times it was the
33	28	0	was the
34	3	7	was the best of times it was the

index(9)

lcp(20)

rank("th")

select(9)

интервалы, содержащие "`th`" и найденные с помощью вызова `rank()` во время бинарного поиска

***Рис. 6.3.4.** Бинарный поиск в суффиксном массиве*

API и код клиента

Для поддержки клиентского кода, решающего эти две задачи, мы сформулируем API, приведенный на рис. 6.3.5. Он содержит конструктор; метод `length()`; методы `select()` и `index()`, которые выдают строку и индекс суффикса заданного ранга в упорядоченном списке суффиксов; метод `lcp()`, который выдает длину максимального общего префикса для каждого суффикса и предшествующего ему в упорядоченном списке; и метод `rank()`, который выдает количество суффиксов, меньших указанного ключа (как при первом нашем знакомстве с бинарным поиском в главе 1). Для описания абстракции упорядоченного списка суффиксных строк мы будем использовать термин *суффиксный массив*, хотя он не обязан быть основан на массиве строк.

public class **SuffixArray**		
	SuffixArray(String text)	*создание суффиксного массива для строки* text
int	length()	*длина* text
String	select(int i)	*i-й элемент в суффиксном массиве (*i *от* 0 *до* N-1*)*
int	index(int i)	*индекс* select(i) *(*i *от* 0 *до* N-1*)*
int	lcp(int i)	*длина максимального общего префикса у* select(i) *и* select(i-1) *(*i *от* 0 *до* N-1*)*
int	rank(String key)	*количество суффиксов, меньших* key

***Рис. 6.3.5.** API для суффиксного массива*

В примере на рис. 6.3.4 вызов `select(9)` дает `"as the best of times..."`, `index(9)` равно 4, а `lcp(20)` равно 10, т.к. подстроки `"it was the best of times..."` и `"it was the"` имеют общий префикс `"it was the"` длиной 10, а `rank("th")` равно 30. Обратите также внимание, что вызов `select(rank(key))` выдает первый возможный суффикс в упорядоченном списке суффиксов, который имеет в качестве префикса строку `key`, а за ним следуют все остальные вхождения `key` в тексте (см. рис. 6.3.4).

При наличии данного API легко написать клиентский код, приведенный в листингах 6.3.2 и 6.3.3. Класс `LRS` (листинг 6.3.2) находит максимальную повторяющуюся подстроку в тексте из стандартного ввода. Для этого он строит суффиксный массив и просматривает упорядоченные суффиксы, чтобы найти максимальное значение `lcp()`. Класс `KWIC` (листинг 6.3.3) строит индексный массив для текста, имя которого задается в аргументе командной строки, принимает запросы из стандартного ввода и выводит все вхождения каждого запроса в тексте (с заданным количеством символов перед ним и после него для обеспечения контекста). Идентификатор `KWIC` означает термин *keyword-in-context* (ключевое слово в контексте), который появился не позднее 1960-х годов. Простота и эффективность этого клиентского пода для типичных приложений обработки строк впечатляет и свидетельствует о важности тщательного проектирования API (и мощи простой, но полезной идеи).

Листинг 6.3.2. Клиент поиска максимальной повторяющейся подстроки

```
public class LRS
{
```

```
   public static void main(String[] args)
   {
      String text = StdIn.readAll();
      int N = text.length();
      SuffixArray sa = new SuffixArray(text);
      String lrs = "";
      for (int i = 1; i < N; i++)
      {
         int length = sa.lcp(i);
         if (length > substring.length())
            lrs = sa.select(i).substring(0, length);
      }
      StdOut.println(lrs);
   }
}
```

```
% more tinyTale.txt
it was the best of times it was the worst of times
it was the age of wisdom it was the age of foolishness
it was the epoch of belief it was the epoch of incredulity
it was the season of light it was the season of darkness
it was the spring of hope it was the winter of despair

% java LRS < tinyTale.txt
st of times it was the
```

Листинг 6.3.3. Клиент индексации ключевых слов в контексте

```
public class KWIC
{
   public static void main(String[] args)
   {
      In in = new In(args[0]);
      int context = Integer.parseInt(args[1]);

      String text = in.readAll().replaceAll("\\s+", " ");;
      int N = text.length();
      SuffixArray sa = new SuffixArray(text);

      while (StdIn.hasNextLine())
      {
         String q = StdIn.readLine();
         for (int i = sa.rank(q); i < N && sa.select(i).startsWith(q); i++)
         {
            int from = Math.max(0, sa.index(i) - context);
            int to   = Math.min(N-1, from + q.length() + 2*context);
            StdOut.println(text.substring(from, to));
         }
         StdOut.println();
      }
   }
}
```

```
% java KWIC tale.txt 15
search
o st giless to search for contraband
her unavailing search for your fathe
le and gone in search of her husband
t provinces in search of impoverishe
 dispersing in search of other carri
n that bed and search the straw hold

better thing
t is a far far better thing that i do than
 some sense of better things else forgotte
was capable of better things mr carton ent
```

Реализация

Код в листинге 6.3.4 представляет собой естественную реализацию API `SuffixArray`. Переменные экземпляров — массив строк и (для экономии в коде) переменная `N`, которая содержит длину массива (длину строки и количество ее суффиксов). Конструктор создает суффиксный массив и сортирует его, после чего вызов `select(i)` просто возвращает `suffixes[i]`. Реализация метода `index()` также умещается в одну строку, но в ней используется маленькая хитрость — наблюдение, что *длина суффиксной строки уникально определяет ее начало в тексте*. Суффикс длиной `N` начинается в позиции 0, суффикс длиной `N-1` начинается в позиции 1, суффикс длиной `N-2` начинается в позиции 2 и т.д., поэтому вызов `index(i)` просто возвращает `N-suffixes[i].length()`. При наличии статического метода `lcp()` (листинг 6.3.1) реализация метода `lcp()` очевидна, а метод `rank()` практически совпадает с реализацией для бинарного поиска в таблице имен (листинг 3.1.7). Как обычно, простота и элегантность этой реализации не должна заслонять тот факт, что это непростой алгоритм, позволяющий решать важную задачу — поиск максимальной повторяющейся подстроки — которую иначе было бы невозможно решить.

Листинг 6.3.4. Алгоритм 6.2. Суффиксный массив (элементарная реализация)

```
public class SuffixArray
{
   private final String[] suffixes;    // суффиксный массив
   private final int N;                // длина строки (и массива)

   public SuffixArray(String s)
   {
      N = s.length();
      suffixes = new String[N];
      for (int i = 0; i < N; i++)
         suffixes[i] = s.substring(i);
      Quick3way.sort(suffixes);
   }

   public int length()          { return N; }
   public String select(int i) { return suffixes[i]; }
   public int index(int i)     { return N - suffixes[i].length(); }
```

```
    private static int lcp(String s, String t)
    // См. листинг 6.3.1.

    public int lcp(int i)
    {  return lcp(suffixes[i], suffixes[i-1]); }

    public int rank(String key)
    {  // бинарный поиск
       int lo = 0, hi = N - 1;
       while (lo <= hi)
       {
          int mid = lo + (hi - lo) / 2;
          int cmp = key.compareTo(suffixes[mid]);
          if      (cmp < 0) hi = mid - 1;
          else if (cmp > 0) lo = mid + 1;
          else return mid;
       }
       return lo;
    }
}
```

Эффективность этой реализации API `SuffixArray` зависит от неизменяемости значений `String` в Java: подстроки представляют собой ссылки одинакового размера, и поэтому извлечение подстроки занимает постоянное время (см. текст).

Производительность

Эффективность сортировки суффиксов зависит от того, что извлечение подстроки в Java выполняется за постоянное время: каждая строка состоит из стандартной информации об объекте, указателя на начало строки и ее длины. Поэтому размер индекса линейно зависит от размера строки. Это с виду противоречит тому, что общее количество символов в суффиксах равно $\sim N^2/2$, т.е. квадратичной функции от длины строки. Кроме того, эта квадратичность несколько обескураживает и при оценке стоимости сортировки суффиксного массива. Очень важно понимать, что этот подход эффективен для длинных строк в силу реализации строк в Java: при обмене двух строк выполняется обмен только ссылок на них, но не самих строк. Но стоимость *сравнения* двух строк может быть пропорциональна длине строк — при наличии очень длинных общих префиксов, однако в типичных приложениях в сравнениях участвуют лишь несколько символов. В этом случае время сортировки суффиксов линейно-логарифмично. Например, многие приложения вполне адекватно описываются моделью случайных строк:

Утверждение В. Используя 3-частную быструю сортировку строк, можно построить суффиксный массив из случайной строки длиной N в памяти с объемом, пропорциональным N, и выполнив в среднем $\sim 2N \ln N$ сравнений символов.

Обсуждение. Граница для объема памяти очевидна, а граница для времени выполнения следует из результата тщательного и сложного исследования П. Жаке (P. Jacquet) и В. Шпанковского (W. Szpankowski), которое показывает, что стоимость сортировки суффиксов асимптотически совпадает со стоимостью сортировки N случайных строк (см. утверждение Д в разделе 5.1).

Усовершенствованные реализации

У нашей элементарной реализации `SuffixArray` плохая производительность в худшем случае. Например, если все символы в тексте одинаковы, сортировка просматривает каждый символ в каждой подстроке и поэтому выполняется за *квадратичное* время. Для строк, которые мы брали в качестве примеров — такие как геномные последовательности или текст на естественном языке — это вряд ли приведет к проблемам, но такой алгоритм может выполняться медленно для текстов с длинными фрагментами из одинаковых символов. На данную проблему можно взглянуть и по-другому: стоимость поиска максимальной повторяющейся подстроки *квадратична относительно длины подстроки*, т.к. необходимо проверить все префиксы повтора (рис. 6.3.6). Это не проблема для текста наподобие "Повести о двух городах", где самый длинный повтор

```
"s dropped because it would have been a bad thing for me in a worldly point of view i"
```

содержит лишь 84 символа, но это выливается в серьезную проблему для геномных данных, где длинные повторяющиеся подстроки встречаются часто. Как можно избежать подобного квадратичного поведения при поиске повторов?

Входная строка

a **a c a a g** t t t **a c a a g** c

Суффиксы самого длинного повтора (M = 5)

```
a c a a g
  c a a g
    a a g
      a g
        g
```

все выступают в качестве префикса суффиксной строки как минимум дважды

Упорядоченные суффиксы входной строки

```
  a a c a a g t t t a c a a g c
3 a a g c
  a a g t t t a c a a g c
5 a c a a g c
  a c a a g t t t a c a a g c
2 a g c
  a g t t t a c a a g c
  c
4 c a a g c
  c a a g t t t a c a a g c
1 g c
  g t t t a c a a g c
  t a c a a g c
  t t a c a a g c
  t t t a c a a g c
```

стоимость сравнения не менее $1+2+...+M \sim M^2/2$

Рис. 6.3.6. *Стоимость максимальной повторяющейся подстроки квадратична относительно длины повтора*

П. Вайнер (P. Weiner) в 1973 г. показал, что *задачу поиска максимальной повторяющейся подстроки можно гарантированно решить за линейное время*. Алгоритм Вайнера основан на построении дерева суффиксов (по сути, это trie-дерево для суффиксов). Однако суффиксные деревья содержат несколько указателей на каждый символ и поэтому занимают слишком много памяти для большинства практических задач, которые приводят к использованию суффиксных массивов. В 1990-х годах Ю. Манбер (U. Manber) и И. Майерс (E. Myers) представили линейно-логарифмический алгоритм для непосред-

ственного построения суффиксных массивов, а также метод, выполняющий предварительную обработку за то же время, что и сортировка суффиксов, и поддерживающий выполнение lcp() за *постоянное время*. С тех пор было разработано еще несколько алгоритмов сортировки суффиксов за линейное время. Если немного постараться, то реализация Манбера-Майерса может поддерживать и метод lcp() с двумя аргументами, который находит максимальный общий префикс двух заданных не обязательно соседних суффиксов за гарантированно постоянное время — еще одно замечательное усовершенствование по сравнению с примитивной реализацией. Эти результаты удивляют: ведь они позволяют получить эффективность далеко за пределами ожидаемой.

Утверждение Г. Суффиксные массивы позволяют решить задачи сортировки суффиксов и поиска максимальной повторяющейся подстроки за линейное время.

Доказательство. Алгоритмы для решения этих задач далеко выходят за рамки данной книги, но на сайте книги приведена реализация конструктора SuffixArray, которая отрабатывает за линейное время, и метода lcp(), который выполняется за постоянное время.

Реализация класса SuffixArray, основанная на этих идеях, поддерживает эффективные решения многочисленных задач обработки строк с помощью простого клиентского кода, как в наших примерах с применением LRS и KWIC.

Суффиксные массивы — результат многолетних исследований, которые начались с разработки trie-деревьев для KWIC-индексов в 1960-х годах. Рассмотренные нами алгоритмы созданы многими исследователями на протяжении нескольких десятилетий в контексте решения практических задач — от перевода в онлайновый режим Оксфордского словаря английского языка до первых поисковых механизмов в веб-сети и исследования генокода человека. Эта история помогает нам лучше уяснить практическую важность построения и анализа алгоритмов.

Упражнения

6.3.1. Приведите в стиле рис. 6.3.6 таблицы суффиксов, упорядоченных суффиксов, index () и lcp() для следующих строк:

а) abacadaba

б) mississippi

в) abcdefghij

г) aaaaaaaaaa

6.3.2. Укажите, в чем проблема с данным кодовым фрагментом, который должен вычислять все суффиксы для последующей сортировки:

```
suffix = "";
for (int i = s.length() - 1; i >= 0; i--)
{
  suffix = s.charAt(i) + suffix;
  suffixes[i] = suffix;
}
```

Ответ: он требует квадратичного объема памяти и квадратичного времени.

6.3.3. В некоторых приложениях требуется выполнять *циклические перестановки* текста, которые содержат все символы текста. *i*-я циклическая перестановка текста длиной N (для i от 0 до $N-1$) представляет собой последние $N-i$ символы текста, за которыми следуют первые i символов. Укажите, в чем проблема со следующим кодовым фрагментом, который должен вычислять все циклические перестановки:

```
int N = s.length();
for (int i = 0; i < N; i++)
   rotation[i] = s.substring(i, N) + s.substring(0, i);
```

Ответ: он требует квадратичного объема памяти и квадратичного времени.

6.3.4. Предложите линейный алгоритм для вычисления всех циклических перестановок текстовой строки.

Ответ:

```
String t = s + s;
int N = s.length();
for (int i = 0; i < N; i++)
  rotation[i] = r.substring(i, i + N);
```

6.3.5. При условиях, описанных в разделе 1.4, приведите объем памяти, требуемой объектом `SuffixArray` для обработки строки длиной N.

6.3.6. *Максимальная общая подстрока.* Напишите клиент `LCS` класса `SuffixArray`, который принимает два имени файлов в качестве аргументов командной строки, читает два текстовых файла и находит за линейное время самую длинную подстроку, которая присутствует в обоих файлах. (В 1970 г. Д. Кнут предположил, что эту задачу решить невозможно.) *Совет*: создайте суффиксный массив для `s#t`, где `s` и `t` — две текстовые строки, а `#` — символ, которого нет ни в одной из них.

6.3.7. *Преобразование Барроуза-Уилера.* Преобразование *Барроуза-Уилера* (Burrows-Wheeler transform — BWT) представляет собой преобразование, которое используется в алгоритмах сжатия данных, в том числе `bzip2`, и в мощных алгоритмах упорядочения в геномике. Напишите клиент класса `SuffixArray`, который вычисляет BWT за линейное время следующим образом. Пусть имеется строка длиной N, оканчивающаяся специальным символом `$`, меньшим всех остальных символов, и матрица $N \times N$, каждая строка которой содержит отличную от других циклическую перестановку исходной текстовой строки. Упорядочьте строки матрицы лексикографически. Преобразование Барроуза-Уилера — это самый правый столбец в упорядоченной матрице. Например, BWT для строки `mississippi$` равно `ipssm$pissii`. Обратное преобразование Барроуза-Уилера (Burrows-Wheeler inverse transform — BWI) выполняет обращение BWT. Например, BWI для строки `ipssm$pissii` равно `mississippi$`. Напишите также клиент, который по заданному BWT для текстовой строки вычисляет BWI за линейное время.

6.3.8. *Циклическая линеаризация строки.* Напишите клиент класса `SuffixArray`, который за линейное время находит для заданной строки ее лексикографически наименьшую циклическую перестановку. Эта задача возникает в химических базах данных для циклических молекул, которые представляются циклическими

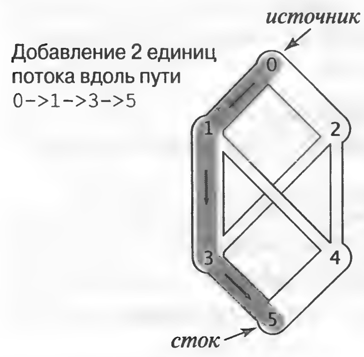
строками. Для поиска по произвольной перестановке в качестве ключа используется каноническое представление (наименьшая циклическая перестановка). (См. упражнение 6.27 и упражнение 6.28.)

6.3.9. *Максимальная подстрока с k повторами*. Напишите клиент класса `SuffixArray`, который для заданной строки и целого числа k находит самую длинную подстроку, которая повторяется в строке k или более раз.

6.3.10. *Длинные повторяющиеся подстроки*. Напишите клиент класса `SuffixArray`, который для заданной строки и целого числа L находит все повторяющиеся подстроки длиной L и более.

6.3.11. *Подсчет частоты k-грамм*. Разработайте и реализуйте АТД для предварительной обработки строки, чтобы затем можно было эффективно отвечать на запросы вида "*Сколько раз присутствует в тексте заданная k-грамма?*" Ответ на каждый запрос должен быть найден в худшем случае за время, пропорциональное $k \log N$, где N — длина строки.

6.4. Алгоритмы для сетевых потоков



Добавление 1 единицы
потока вдоль пути
0->2->4->5





Рис. 6.4.1. *Добавление потока в сеть*

Сейчас мы рассмотрим модель графа, которая доказала свою успешность не только потому, что она просто формулируется и полезна во многих практических ситуациях, но и потому, что имеются эффективные алгоритмы решения задач в рамках этой модели. Решение, которое мы рассмотрим, иллюстрирует компромисс между стремлением к реализациям общего назначения и стремлением находить эффективные решения для конкретных задач. Изучение алгоритмов для сетевых потоков интересно и само по себе, поскольку оно очень близко подводит нас к компактным и элегантным реализациям, которые достигают обеих целей. Как вы увидите, имеются очевидные реализации, которые гарантированно отрабатывают за время, пропорциональное полиному от размера сети.

Классические решения задач для сетевых потоков тесно связаны с другими алгоритмами на графах, с которыми мы познакомились в главе 4, и мы можем написать на удивление лаконичные программы для их решения, используя разработанные нами алгоритмические инструменты. Как мы уже не раз убеждались в других ситуациях, хорошие алгоритмы и структуры данных могут существенно снизить время решения задач. Разработка лучших реализаций и лучших алгоритмов постоянно является областью активного поиска, и постоянно открываются все новые подходы и способы.

Физическая модель

Сначала мы рассмотрим идеализированную физическую модель, в которой естественно вводятся несколько базовых концепций. А именно, представьте себе набор трубопроводов различных размеров, которые соединены между собой и содержат вентили, управляющие направлением потоков в местах соединений, как на рис. 6.4.1. Предположим также, что в такой сети имеется один *источник* (например, нефтяная вышка) и один *сток* (например, нефтеперерабатывающее предприятие), с которыми, в конечном счете, соединены все трубы. В каждом месте соединения труб протекающие по ним потоки сбалансированы: количество втекающей жидкости равно количеству вытекающей. Потоки и пропускная способность труб измеряются в одинако-

вых единицах (например, в литрах в секунду). Если в каждом соединении общая пропускная способность входящих труб равна общей пропускной способности выходящих труб, то нечего и решать: нужно просто полностью заполнить все трубы. Иначе не все трубы будут заполнены по максимуму, но жидкость все-таки будет протекать по трубам, и этим течением можно управлять с помощью вентилей в местах соединений, соблюдая условие *локального баланса* (рис. 6.4.2): количество жидкости, втекающей в каждое соединение, равно количеству вытекающей оттуда жидкости. Например, рассмотрим диаграмму на рис. 6.4.1. Операторы могут сначала пустить поток, открыв вентили на пути 0->1->3->5, который может дать мощность в 2 единицы, затем открыть вентили по пути 0->2->4->5, что увеличит поток еще на единицу. Поскольку трубы 0->1, 2->4 и 3->5 уже заполнены, нет прямого способа передать больший поток из 0 в 5, но если с помощью вентиля 1 перенаправить дополнительный поток в 1->4, то появится дополнительная пропускная способность в 3->5, которая позволит увеличить поток 0->2->3->5 еще на единицу.



Рис. 6.4.2. Локальный баланс в транспортной сети

Даже в такой простой сети поиск настроек вентилей, которые увеличивают поток, является весьма непростой задачей. Поэтому понятно, что для сложных сетей нас интересует следующий вопрос: какие параметры вентилей помогут пропустить максимальное количество жидкости из источника в сток? Эту ситуацию можно смоделировать непосредственно с помощью графа с взвешенными ребрами (сети), одним источником и одним стоком (рис. 6.4.3). Ребра сети соответствуют трубопроводам, вершины — вентилям, которые управляют величинами потоков, проходящих в каждое исходящее ребро, а веса ребер — пропускной способности труб. Предполагается, что ребра являются направленными, т.е. жидкость может протекать по трубам только в одном направлении. Каждая труба может содержать поток определенной мощности, который меньше или равен ее пропускной способности, и в каждой вершине выполняется условие баланса: приток равен оттоку.

Эта абстракция транспортной сети представляет собой полезную модель для решения задач, которая непосредственно применима ко многим различными приложениям, а косвенно к еще большему количеству.

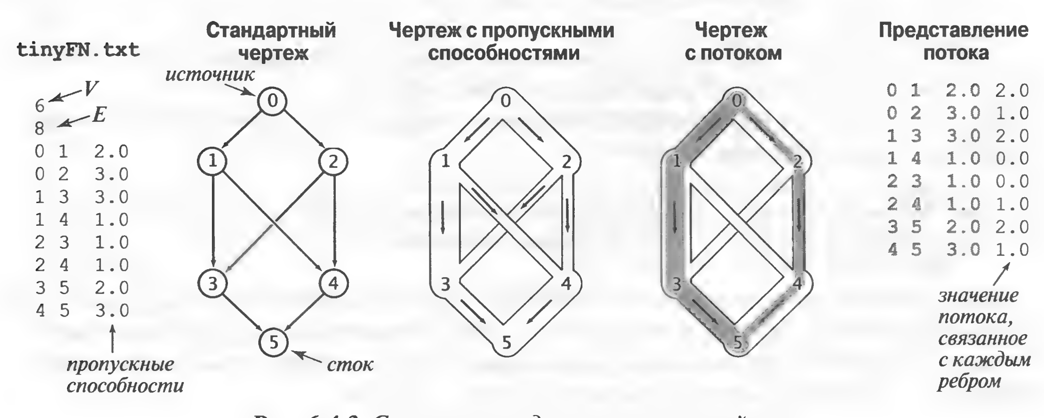


Рис. 6.4.3. Структура задачи транспортной сети

Иногда мы будем возвращаться к физической схеме протекания жидкости по трубопроводам, чтобы иллюстрировать базовые понятия, но наше обсуждение не менее хорошо описывает и товары, перевозимые по каналам распространения, и множество других ситуаций. Как и в случае использования расстояния в алгоритмах поиска кратчайших путей, мы в любой момент можем отбросить все физические аналогии, т.к. все рассматриваемые определения, свойства и алгоритмы основаны только на абстрактной модели, которая не обязательно должна согласовываться с физическими законами. И мы интересуемся моделью сетевых потоков главным образом потому, что, как будет показано в следующем разделе, она позволяет решать с помощью сведения многочисленные другие задачи.

Определения

В силу широкой применимости имеет смысл точно сформулировать термины и концепции, которые мы только что ввели неформально.

Определение. *Транспортная сеть* — это взвешенный орграф, ребра которого имеют положительные веса (которые мы будем называть *пропускными способностями*). *st-сеть* содержит две выделенные вершины — источник s и сток t.

Иногда про ребра говорят, что они имеют бесконечную пропускную способность или, что эквивалентно, являются безразмерными. Это означает, что мы не сравниваем поток с пропускной способностью таких ребер — либо же можно использовать сигнальное значение, которое гарантированно больше, чем любое значение потока. Общий поток в вершину (сумма потоков по всем входящим в нее ребрам) мы будем называть *притоком* этой вершины, общий поток из вершины (сумма потоков по всем исходящим из нее ребрам) мы будем называть *оттоком* этой вершины, а разность притока и оттока — *результирующим потоком*. Для простоты мы будем предполагать, что в сети нет ребер, выходящих из t или входящих в s.

Определение. *st-поток* в *st-сети* — это множество неотрицательных значений, связанных с каждым ребром, которые называются *реберными потоками*. Мы будем называть поток *допустимым*, если он удовлетворяет условию, что ни в одном ребре поток не превышает пропускной способности этого ребра, и результирующий поток в каждой вершине равен нулю (кроме s и t).

Приток в сток мы будем называть *значением st*-потока. В утверждении Д будет показано, что это значение также равно оттоку из источника.

С учетом всех этих определений формальное описание нашей основной задачи выглядит следующим образом.

Максимальный st-поток. Для заданной *st*-сети нужно найти *st*-поток, такой, что никакой другой поток из s в t имеет большее значение.

Для краткости мы будем называть такой поток *максимальным потоком*, а задачу его нахождения в сети — *задачей максимального потока*. В некоторых случаях бывает достаточно знать значение максимального потока, но обычно требуется знать поток (значения потоков в ребрах), который приводит к этому значению.

API

API FlowEdge и FlowNetwork, приведенные на рис. 6.4.4 и 6.4.5, представляют собой естественные расширения API из главы 3. В листинге 6.4.2 будет представлена реализация FlowEdge, основанная на добавлении в класс WeightedEdge (рис. 4.3.1) переменной экземпляров, которая содержит значение потока (рис. 6.4.6). Потоки имеют направления, но мы не будем порождать класс FlowEdge от класса WeightedDirectedEdge, потому что будем работать с более общей абстракцией — *остаточной сетью*, которая описывается ниже, а для реализации остаточных сетей нужно, чтобы каждое ребро присутствовало в списках смежности обеих его вершин. Остаточные сети позволяют добавлять и вычитать потоки и проверять, заполнено ли какое-то ребро до отказа (невозможно добавить никакой поток) или пусто (невозможно вычесть никакой поток). Эта абстракция реализуется с помощью методов residualCapacity() и addResidualFlow(), которые будут описаны ниже.

public class **FlowEdge**		
	FlowEdge(int v, int w, double cap)	
int	from()	*вершина, из которой исходит данное ребро*
int	to()	*вершина, на которое указывает данное ребро*
int	other(int v)	*другая вершина ребра*
double	capacity()	*пропускная способность ребра*
double	flow()	*поток в данном ребре*
double	residualCapacityTo(int v)	*остаточная пропускная способность в направлении* v
double	addFlowTo(int v, double delta)	*добавление потока* delta *в направлении* **v**
String	toString()	*строковое представление*

Рис. 6.4.4. *API для ребер в транспортной сети*

public class **FlowNetwork**		
	FlowNetwork(int V)	*пустая транспортная сеть с V вершинами*
	FlowNetwork(In in)	*построение сети из входного потока*
int	V()	*количество вершин*
int	E()	*количество ребер*
void	addEdge(FlowEdge e)	*добавление ребра* e *в транспортную сеть*
Iterable<FlowEdge>	adj(int v)	*ребра, исходящие из вершины* v
Iterable<FlowEdge>	edges()	*все ребра данной транспортной сети*
String	toString()	*строковое представление*

Рис. 6.4.5. *API для транспортной сети*

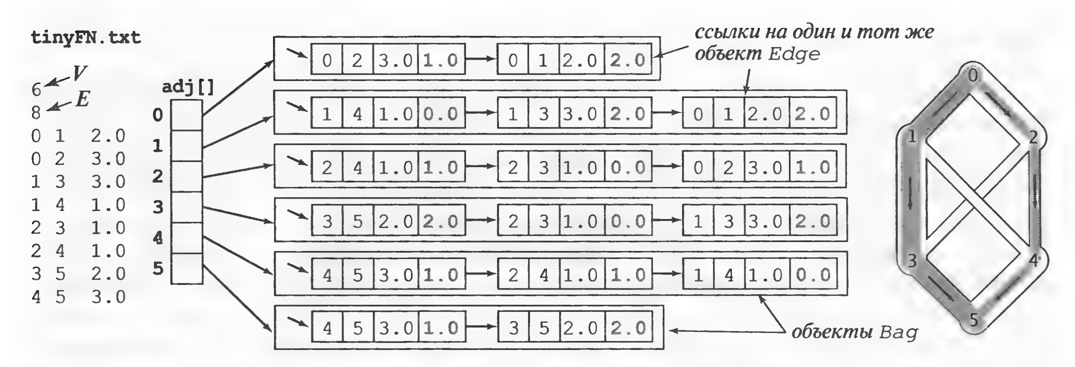


Рис. 6.4.6. Представление транспортной сети

Реализация класса `FlowNetwork` практически идентична нашей реализации `EdgeWeightedGraph` из листинга 4.3.3, поэтому мы не приводим ее. Для упрощения формата файлов мы будем придерживаться соглашения, что источником является вершина 0, а стоком — вершина $N-1$. Эти API предоставляют поле деятельности для алгоритмов нахождения максимальных потоков: сначала строится транспортная сеть, а затем переменным экземпляров в клиентских ребрах присваиваются значения, которые максимизируют поток через эту сеть. В листинге 6.4.1 приведены клиентские методы для проверки, действительно ли поток является допустимым. Обычно такое действие можно выполнять после отработки алгоритма нахождения максимального потока.

Листинг 6.4.1. Проверка допустимости потока в транспортной сети

```
private boolean localEq(FlowNetwork G, int v)
{  // Проверка локального баланса в каждой вершине.
   double EPSILON = 1E-11;
   double netflow = 0.0;

   for (FlowEdge e : G.adj(v))
      if (v == e.from()) netflow -= e.flow();
      else               netflow += e.flow();
   return Math.abs(netflow) < EPSILON;
}

private boolean isFeasible(FlowNetwork G)
{
   // Проверка, что поток по каждому ребру неотрицателен
   //  и не превышает его пропускную способность.
   for (int v = 0; v < G.V(); v++)
      for (FlowEdge e : G.adj(v))
         if (e.flow() < 0 || e.flow() > e.cap())
            return false;

   // Проверка локального баланса в каждой вершине.
   for (int v = 0; v < G.V(); v++)
      if (v !=s && v != t && !localEq(v))
         return false;
   return true;
}
```

Алгоритм Форда-Фалкерсона

Эффективный способ решения задач нахождения максимального потока был разработан Л.Р. Фордом (L.R. Ford) и Д.Р. Фалкерсоном (D.R. Fulkerson) в 1962 г. Это обобщенный метод пошагового увеличения потоков по путям из источника в сток, на котором основано целое семейство алгоритмов. В классической литературе он называется *алгоритмом Форда-Фалкерсона*, но, кроме этого, широко применяется и более описательный термин *алгоритм расширения пути* (augmenting-path).

Рассмотрим любой направленный путь из источника в сток через *st*-сеть. Пусть *x* — минимальная незадействованная пропускная способность ребер на этом пути. Значение сетевого потока можно увеличить, по крайней мере, на *x*, увеличив на эту величину поток во всех ребрах пути. Повторение этого действия дает первую попытку вычисления потока в сети: находим другой путь, увеличиваем поток по этому пути, и продолжаем так, пока все пути из источника в сток не будут содержать хотя бы одно заполненное ребро (и поэтому поток таким образом увеличить уже нельзя). В некоторых случаях этот алгоритм может вычислять максимальный поток, но во многих других случаях он не приводит к нужному результату. Пример подобных действий приведен на рис. 6.4.1.

Чтобы алгоритм всегда находил максимальный поток, мы рассмотрим более общий способ увеличения потока — вдоль пути в сети из источника в сток по ребрам соответствующего *неориентированного* графа. Ребра на любом таком пути являются либо *прямыми* ребрами, которые совпадают с направлением потока (при проходе по пути из источника в сток мы проходим по ребру из его исходной вершины в конечную), либо *обратными* ребрами, которые противоположны направлению потока (при проходе по пути из источника в сток мы проходим по ребру из конечной вершины в начальную).

Теперь для любого пути с незаполненными прямыми ребрами и непустыми обратными ребрами можно увеличить величину потока в сети, увеличивая поток в прямых ребрах и уменьшая в обратных. Такой путь называется *расширяющим путем*, пример такого пути приведен на рис. 6.4.7. В суммарном потоке, по крайней мере, одно прямое ребро становится заполненным или, по крайней мере, одно обратное ребро становится пустым. Этот схематических описанный процесс лежит в основе классического алгоритма Форда-Фалкерсона для нахождения максимального потока (метод расширяющего пути). Кратко его можно сформулировать следующим образом.



***Рис. 6.4.7.** Расширяющий путь* (0->2->3->1->4->5)

Алгоритм нахождения максимального потока Форда-Фалкерсона. Начинаем с нулевых потоков по всем ребрам. Увеличиваем поток по любому расширяющему пути из источника в сток (в котором нет заполненных прямых ребер и пустых обратных ребер), и повторяем это действие, пока в сети не останется таких путей.

Интересно, что (при некоторых технических условиях, касающихся числовых свойств потока) этот метод всегда находит максимальный поток, независимо от способа выбора путей. Аналогично “жадному” алгоритму нахождения МОД, описанному в разделе 4.3, и обобщенному методу обнаружения кратчайших путей, который рассмотрен в разделе 4.4, это обобщенный алгоритм, который полезен тем, что он обосновывает корректность целого семейства более конкретных алгоритмов. Для выбора пути можно использовать любой метод. Разработано несколько алгоритмов, вычисляющих последовательности расширяющих путей, и все они приводят к получению максимального потока. Эти алгоритмы различаются количеством вычисляемых расширяющих путей и стоимостью нахождения каждого такого пути, но все они реализуют алгоритм Форда-Фалкерсона и находят максимальный поток.

Теорема о максимальном потоке и минимальном сечении

Для доказательства, что любой поток, вычисленный любой реализацией алгоритма Форда-Фалкерсона, действительно является максимальным потоком, мы докажем ключевой факт, который называется *теоремой о максимальном потоке и минимальном сечении* (maxflow-mincut theorem). Понимание этой теоремы крайне важно для понимания работы алгоритмов сетевых потоков. Как следует из ее названия, эта теорема основана на прямой связи между потоками и сечениями в сетях, поэтому мы начнем с определения терминов, относящихся к сечениям. Вспомните из раздела 4.3, что *сечение* в графе представляет собой разбиение вершин на два непересекающихся множества, а перекрестное ребро — это ребро, которое соединяет вершину из одного такого множества с вершиной из другого множества. Для транспортных сетей эти определения можно сформулировать следующим образом.

Определение. *st-сечение* — это сечение, которое помещает вершину s в одно из его множеств, а вершину t — в другое.

Каждое перекрестное ребро, соответствующее *st*-сечению, является либо *st-ребром*, которое направлено из вершины во множестве, содержащем s, в вершину во множестве, содержащем t, или *ts-ребром*, которое направлено в другую сторону. Иногда множество перекрестных *st*-ребер называется *секущим множеством*. *Пропускная способность st*-сечения в транспортной сети равна сумме пропускных способностей *st*-ребер этого сечения, а *поток через st*-сечение равен разности между суммой потоков в его *st*-ребрах и суммой потоков в его *ts*-ребрах. После удаления всех *st*-ребер (секущего множества) из *st*-сечения транспортной сети не остается ни одного пути из s в t, но возврат любого из них создает такой путь.

Сечения представляют собой удобную абстракцию для многих приложений. Для модели с потоками жидкости сечение предоставляет возможность полностью остановить поток жидкости из источника в сток. Если рассматривать пропускную способность сечения как стоимость такой остановки, то для максимально экономичной остановки потока следует решить следующую задачу.

Минимальное st-сечение. Для заданной *st*-сети нужно найти *st*-сечение, такое, что нет никакого другого сечения с меньшей пропускной способностью. Для краткости мы будем называть такое сечение *минимальным сечением* (mincut), а задачу его нахождения в сети — *задачей минимального сечения*.

В описании задачи минимального сечения нет упоминания о потоках, и эти определения могут показаться отходом от нашего обсуждения алгоритма расширения пути. На первый взгляд вычисление минимального сечения (множества ребер) выглядит проще вычисления максимального потока (назначения весов всем ребрам). Однако эти задачи тесно связаны. Доказательство содержится в методе расширения пути. Это доказательство основано на следующей базовой взаимосвязи потоков и сечений, которая непосредственно дает доказательство, что локальный баланс в *st*-потоке и означает глобальный баланс (первое следствие), и дает верхнюю границу значения любого *st*-потока (второе следствие).

Утверждение Д. Для любого *st*-потока поток через произвольное *st*-сечение равен значению потока.

Доказательство. Пусть C_s — множество вершин, содержащее *s*, а C_t — множество вершин, содержащее *t*. Тогда это утверждение непосредственно следует по индукции по размеру C_t (рис. 6.4.8). Оно верно по определению, когда C_t содержит только *t*, и если переместить какую-то вершину из C_s в C_t, то из локального баланса в этой вершине следует сохранение утверждения. Такое перемещение вершин позволяет создать произвольное *st*-сечение.

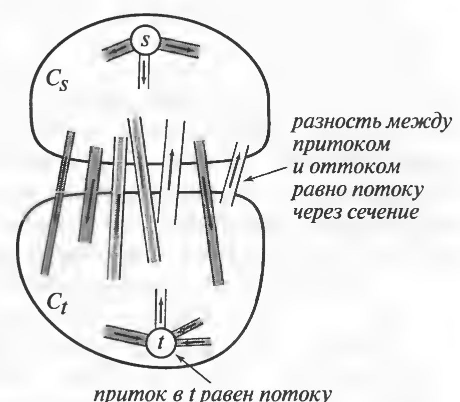


Рис. 6.4.8. К доказательству утверждения Д

Следствие. Отток из *s* равен притоку в *t* (значение *st*-потока).

Доказательство. Пусть C_s равно $\{s\}$.

Следствие. Никакой *st*-поток не может быть больше пропускной способности любого *st*-сечения.

Утверждение Е (теорема о максимальном потоке и минимальном сечении). Пусть имеется *st*-поток *f*. Следующие три условия эквивалентны.

1. Существует *st*-сечение, пропускная способность равна значению потока *f*.
2. *f* — максимальный поток.
3. Не существует путь, расширяющий *f*.

Доказательство. Условие 2 следует из условия 1 согласно следствию из утверждения Д. Условие 3 следует из условия 2, т.к. существование расширяющего пути означает существование потока с большим значением, что противоречит максимальности *f*.

Осталось доказать, что из условия 3 следует условие 1. Пусть C_s — множество всех вершин, достижимых из s по неориентированному пути, который не содержит заполненных прямых и пустых обратных ребер, и пусть C_t — множество остальных вершин. Тогда вершина t должна принадлежать C_t, и, значит, (C_s, C_t) является *st*-сечением, множество которого состоит только из заполненных прямых или пустых обратных ребер. Поток через это сечение равен пропускной способности сечения (поскольку прямые ребра заполнены, а обратные ребра пусты), а также значению потока сети (согласно утверждению Д).

Следствие (свойство целостности). Если пропускные способности выражаются целыми числами, то существует максимальный поток с целочисленным значением, и алгоритм Форда-Фалкерсона находит его.

Доказательство. Каждый расширяющий путь увеличивает значение потока на положительное число (минимум из незадействованных пропускных способностей в прямых ребрах и потоков в обратных ребрах, а все они всегда являются положительными целыми числами).

Можно построить максимальный поток на основе и не целочисленных потоков, даже если все пропускные способности выражаются целыми числами, но мы не будем рассматривать такие потоки. С теоретической точки зрения это наблюдение очень важно: если разрешить вещественные значения пропускных способностей и потоков, как мы сделали, и как обычно делается на практике, то это может привести к неприятным аномальным ситуациям. Например, известно, что алгоритм Форда-Фалкерсона в принципе может привести к бесконечной последовательности расширяющих путей, которая даже не сходится к значению максимального потока. Известно, что версия алгоритма, которую мы рассмотрим, сходится всегда, даже при вещественных значениях пропускных способностей и потоков. Вне зависимости от способа нахождения расширяющего пути и от найденных путей, мы всегда, в конце концов, придем к потоку, для которого нет расширяющего пути — т.е. к максимальному потоку.

Остаточная сеть

Обобщенный алгоритм Форда-Фалкерсона не указывает конкретный метод нахождения расширяющего пути. Как можно найти путь без заполненных прямых ребер и пустых обратных ребер? Ответ на этот вопрос мы начнем со следующего определения.

Определение. Пусть имеется *st*-сеть и *st*-поток. *Остаточная сеть* для этого потока содержит те же вершины, что и исходная, и одно или два ребра для каждого ребра исходной сети, которые определяются следующим образом. Пусть это ребро e из вершины v в вершину w исходной сети, f_e — поток по нему, а c_e — его пропускная способность. Если f_e положительно, в остаточной сети имеется ребро w->v с пропускной способностью f_e, а если f_e меньше c_e, то в остаточной сети имеется ребро v->w с пропускной способностью $c_e - f_e$.

Если исходное ребро e из вершины v в вершину w пусто (f_e равно 0), то в остаточной сети имеется единственное соответствующее ребро v->w с пропускной способностью f_e. Если это ребро заполнено (f_e равно c_e), то в остаточной сети имеется единственное соответствующее ребро w->v с пропускной способностью f_e. А если оно не пусто и не заполнено, то в остаточной сети присутствуют оба ребра v->w и w->v с соответствующими пропускными способностями. Пример такой сети приведен на рис. 6.4.9.

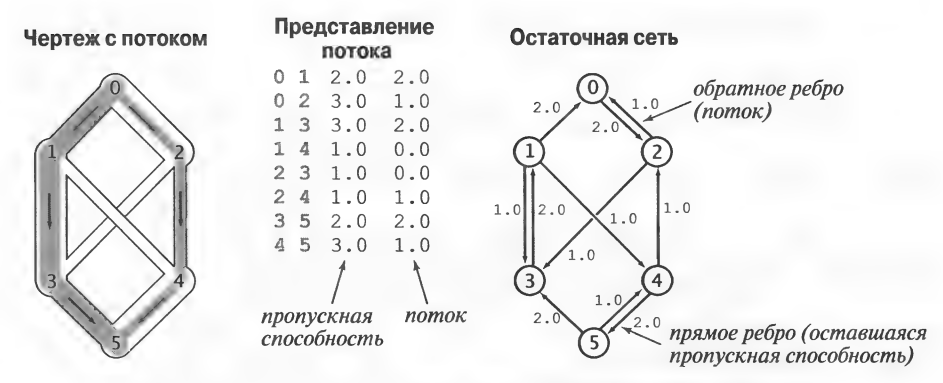


Рис. 6.4.9. Анатомия задачи транспортной сети (еще раз)

На первый взгляд представление остаточной сети выглядит несколько странно, поскольку ребра, соответствующие потокам, направлены *в обратную сторону* по отношению к самим потокам. Прямые ребра представляют оставшуюся пропускную способность (величину потока, который можно добавить в это ребро), а обратные ребра представляют поток (величину потока, который можно изъять из этого ребра).

Код в листинге 6.4.2 содержит методы класса `FlowEdge`, которые нужны для реализации абстракции остаточной сети. Работая с этими реализациями, наши алгоритмы работают с остаточной сетью, но в реальности они проверяют пропускные способности и изменяют потоки (с помощью ссылок на ребра) в ребрах клиентской сети. Методы `from()` и `other()` позволяют работать с ребрами в любом направлении: вызов `e.other(v)` возвращает вершину ребра `e`, отличную от `v`. Методы `residualCapTo()` и `addResidualFlowTo()` реализуют остаточную сеть.

Листинг 6.4.2. Тип данных для ребра транспортной сети (остаточная сеть)

```
public class FlowEdge
{
   private final int v;                        // начальная вершина ребра
   private final int w;                        // конечная вершина ребра
   private final double capacity;              // пропускная способность
   private double flow;                        // поток

   public FlowEdge(int v, int w, double capacity)
   {
      this.v = v;
      this.w = w;
      this.capacity = capacity;
      this.flow = 0.0;
   }

   public int from()             {  return v;          }
   public int to()               {  return w;          }
   public double capacity()      {  return capacity;   }
   public double flow()          {  return flow;       }

   public int other(int vertex)
   // такой же, как для Edge
```

```
  public double residualCapacityTo(int vertex)
  {
     if      (vertex == v) return flow;
     else if (vertex == w) return cap - flow;
     else throw new RuntimeException("Недопустимое ребро");
  }
  public void addResidualFlowTo(int vertex, double delta)
  {
     if      (vertex == v) flow -= delta;
     else if (vertex == w) flow += delta;
     else throw new RuntimeException("Недопустимое ребро");
  }
  public String toString()
  {  return String.format("%d->%d %.2f %.2f", v, w, capacity, flow);  }
}
```

Эта реализация FlowEdge добавляет во взвешенную реализацию DirectedEdge из раздела 4.4 (см. листинг 4.4.1) переменную экземпляров flow и два метода для реализации остаточной транспортной сети.

Остаточные сети позволяют использовать поиск на графе, чтобы найти расширяющий путь, т.к. любой путь из источника в сток в остаточной сети непосредственно соответствует расширяющему пути в исходной сети. При увеличении потока по этому пути остаточная сеть меняется — например, по крайней мере, одно ребро на пути становится заполненным или пустым — и поэтому, по крайней мере, одно ребро в остаточной сети меняет направление или пропадает (но наше использование абстрактной остаточной сети означает, что мы просто находим положительные пропускные способности, но не вставляем или удаляем ребра).

Метод кратчайшего расширяющего пути

Возможно, самая простая реализация алгоритма Форда-Фалкерсона основана на использовании *кратчайшего* расширяющего пути (в смысле количества ребер на пути, а не его потока или пропускной способности). Этот метод был предложен Дж. Эдмондсом (J. Edmonds) и Р. Карпом (R. Karp) в 1972 г. В нем поиск расширяющего пути выполняется с помощью поиска в ширину (ПвШ) в остаточной сети — в точности так, как описано в разделе 4.1, в чем можно убедиться, сравнивая реализацию метода hasAugmentingPath() (листинг 6.4.3) с нашей реализацией поиска в ширину в алгоритме 4.2 (листинг 4.1.6). Данный метод лежит в основе полной реализации в алгоритме 6.3 (листинг 6.4.4) — это весьма лаконичная реализация на основе разработанных нами инструментов. Для краткости мы будем называть этот метод алгоритмом нахождения максимального потока с помощью *кратчайшего расширяющего пути*. Подробная трассировка для нашего примера приведена на рис. 6.4.10.

Листинг 6.4.3. Нахождение расширяющего пути в остаточной сети с помощью поиска в ширину

```
private boolean hasAugmentingPath(FlowNetwork G, int s, int t)
{
   marked = new boolean[G.V()];          // Известен ли путь к этой вершине?
   edgeTo = new FlowEdge[G.V()];         // Последнее ребро в пути
   Queue<Integer> q = new Queue<Integer>();
```

```
      marked[s] = true;                    // Пометка источника
      q.enqueue(s);                        //   и занесение ее в очередь.
      while (!q.isEmpty())
      {
         int v = q.dequeue();
         for (FlowEdge e : G.adj(v))
         {
            int w = e.other(v);
            if (e.residualCapacityTo(w) > 0 && !marked[w])
            {  // Для каждого ребра к непомеченным вершинам (в остаточной сети)
               edgeTo[w] = e;              // Сохранение последнего ребра в пути.
               marked[w] = true;           // Пометка w, т.к. путь известен,
               q.enqueue(w);               //   и занесение ее в очередь.
            }
         }
      }
      return marked[t];
   }
```

Листинг 6.4.4. Алгоритм 6.3. Алгоритм Форда–Фалкерсона поиска кратчайшего расширяющего пути

```
public class FordFulkerson
{
   private boolean[] marked;   // Принадлежит ли путь s->v остаточному графу?
   private FlowEdge[] edgeTo;  // Последнее ребро в кратчайшем пути s->v
   private double value;       // Текущее значение максимального потока

   public FordFulkerson(FlowNetwork G, int s, int t)
   {  // Нахождение максимального потока в транспортной сети G из s в t.

      while (hasAugmentingPath(G, s, t))
      {  // Пока существует расширяющий путь, используем его.

         // Вычисление минимальной пропускной способности.
         double bottle = Double.POSITIVE_INFINITY;
         for (int v = t; v != s; v = edgeTo[v].other(v))
            bottle = Math.min(bottle, edgeTo[v].residualCapacityTo(v));

         // Расширение пути.
         for (int v = t; v != s; v = edgeTo[v].other(v))
            edgeTo[v].addResidualFlowTo(v, bottle);
         value += bottle;
      }
   }

   public double value()         {  return value;     }
   public boolean inCut(int v)   {  return marked[v]; }

   public static void main(String[] args)
   {
      FlowNetwork G = new FlowNetwork(new In(args[0]));
      int s = 0, t = G.V() - 1;
      FordFulkerson maxflow = new FordFulkerson(G, s, t);
      StdOut.println("Максимальный поток из " + s + " в " + t);
```

```
      for (int v = 0; v < G.V(); v++)
         for (FlowEdge e : G.adj(v))
            if ((v == e.from()) && e.flow() > 0)
               StdOut.println("   " + e);
      StdOut.println("Величина максимального потока = " + maxflow.value());
   }
}
```

Эта реализация алгоритма Форда-Фалкерсона находит кратчайший расширяющий путь в остаточной сети, находит минимальную пропускную способность в этом пути и расширяет поток вдоль этого пути, продолжая так до исчерпания путей из источника в сток.

```
% java FordFulkerson tinyFN.txt
Максимальный поток из 0 в 5
  0->2 3.0 2.0
  0->1 2.0 2.0
  1->4 1.0 1.0
  1->3 3.0 1.0
  2->3 1.0 1.0
  2->4 1.0 1.0
  3->5 2.0 2.0
  4->5 3.0 2.0
Величина максимального потока = 4.0
```

Производительность

Более крупный пример приведен на рис. 6.4.11. Как видно из этого рисунка, длины расширяющих путей образуют неубывающую последовательность. Это наблюдение является первым ключом для анализа производительности алгоритма.

Утверждение Ж. Количество расширяющих путей, необходимых для работы реализации алгоритма максимального потока Форда-Фалкерсона с помощью кратчайших расширяющих путей в транспортной сети с V вершинами и E ребрами, не превышает $EV/2$.

Набросок доказательства. В каждом расширяющем пути имеется *критическое ребро* — ребро, которое будет удалено из остаточной сети, поскольку оно соответствует или прямому ребру, которое будет заполнено до пропускной способности, или обратному ребру, которое станет пустым. Каждый раз, когда ребро является критическим, длина расширяющего пути через него должна увеличиться на 2 (см. упражнение 6.39). Длина расширяющего пути не может превышать V, и поэтому каждое ребро может принадлежать не более чем $V/2$ расширяющим путям, а общее количество расширяющих путей не превышает $EV/2$.

Следствие. Реализация алгоритма нахождения максимального потока Форда-Фалкерсона с помощью кратчайших расширяющих путей в худшем случае выполняется за время, пропорциональное $VE^2/2$.

Доказательство. Поиск в ширину просматривает не более E ребер.

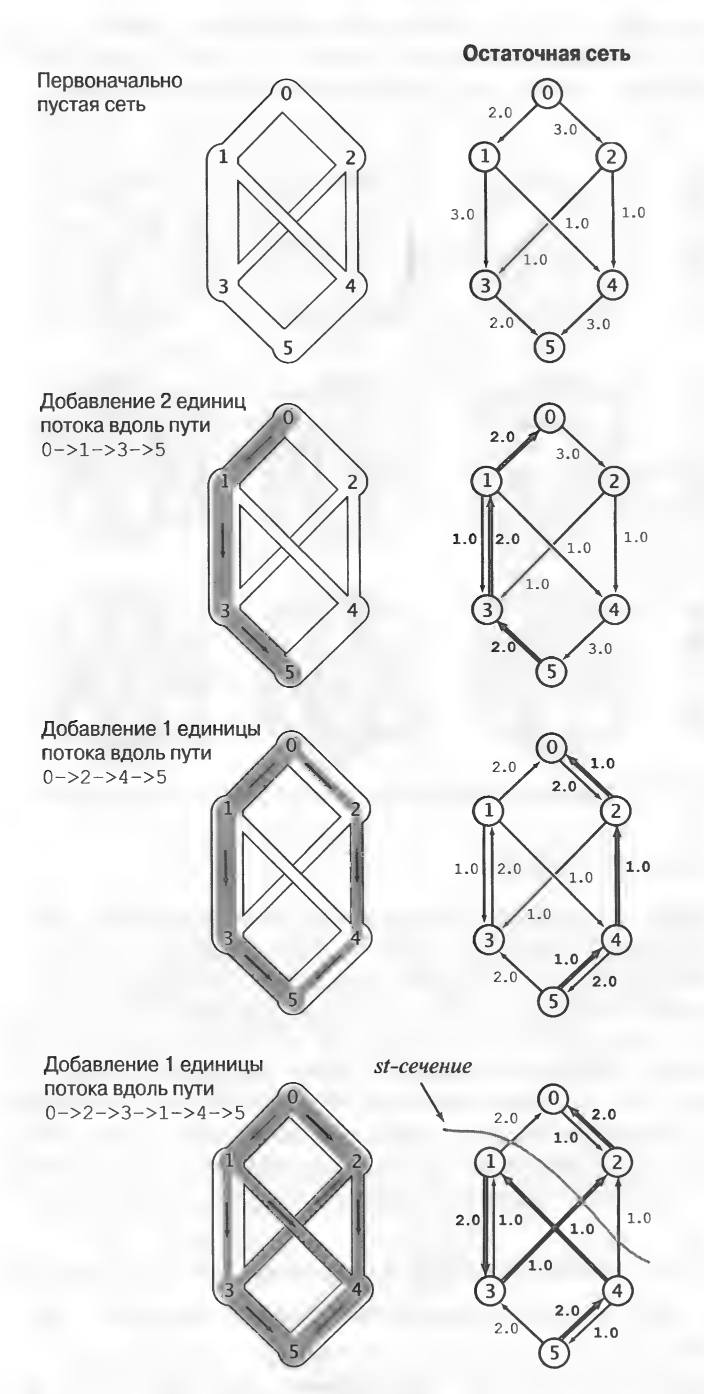


Рис. 6.4.10. Трассировка работы алгоритма расширяющего пути Форда-Фалкерсона

Верхняя граница в утверждении Ж очень осторожна. Например, граф, приведенный на рис. 6.4.11, содержит 11 вершин и 20 ребер, и из этой верхней границы следует, что алгоритм использует не более 110 расширяющих путей. На самом деле он использует лишь 14.

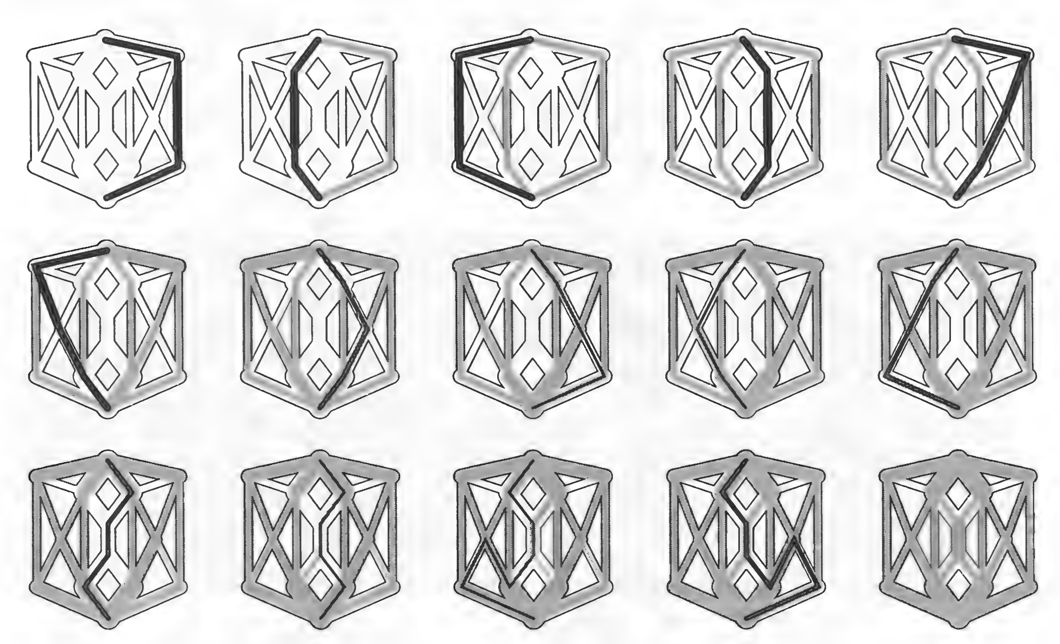


Рис. 6.4.11. Кратчайшие расширяющие пути в более крупной транспортной сети

Другие реализации

Другую реализацию алгоритма Форда-Фалкерсона предложили Эдмондс и Карп: расширение выполняется вдоль пути, который увеличивает поток на максимальную величину. Для краткости этот метод мы будем называть алгоритмом поиска максимального потока с помощью *расширяющего пути максимальной пропускной способности*. Этот подход (и другие) можно реализовать с помощью очереди с приоритетами и слегка измененного алгоритма Дейкстры поиска кратчайших путей: мы выбираем ребра из очереди с приоритетами так, чтобы максимизировать поток, который можно пропустить через прямое ребро или извлечь из обратного ребра. Но можно искать и самый длинный расширяющий путь или выбирать путь случайным образом. Полный анализ для выявления наилучшего метода является сложной задачей, т.к. время выполнения таких алгоритмов зависит от:

- количества расширяющих путей, необходимых для поиска максимального потока;
- времени, необходимого для нахождения каждого расширяющего пути.

Эти величины могут существенно различаться в зависимости от обрабатываемой сети и стратегии поиска на графе. Разработано и несколько других способов решения задачи максимального потока, и некоторые из них успешно конкурируют на практике с алгоритмом Форда-Фалкерсона. Однако разработка математической модели алгоритмов максимального потока, которая может подтвердить такие гипотезы, очень сложна, так

что анализ алгоритмов максимального потока остается интересной и активной областью исследования. С теоретической точки зрения, вычислены границы производительности в худшем случае для многочисленных алгоритмов максимального потока, но они обычно значительно выше, чем стоимости, которые наблюдаются в реальных приложениях, и обычно намного выше, чем тривиальная (линейная) нижняя граница. Этот разрыв между тем, что известно, и тем, что возможно, больше, чем для любых других задач, которые мы рассматривали в данной книге.

Практическое применение алгоритмов максимального потока остается и искусством, и наукой. Искусство состоит в подборе стратегии, наиболее эффективной в данной практической ситуации, а наука — в уяснении сути задачи. И пока неизвестно, существуют ли новые структуры данных и алгоритмы, которые смогут решить задачу максимального потока за линейное время, или мы сможем доказать, что это невозможно.

Таблица 6.4.1. Характеристики производительности алгоритмов нахождения максимального потока

Алгоритм	Порядок величины времени выполнения в худшем случае для V вершин и E ребер с целочисленными пропускными способностями (максимальное = C)
Форда-Фалкерсона с кратчайшими расширяющими путями	VE^2
Форда-Фалкерсона с максимальными расширяющими путями	$E^2 \log C$
Проталкивание предпотока	$EV \log(E/V^2)$
Возможно ли?	$V + E$?

Упражнения

6.4.1. Если пропускные способности представляют собой положительные целые числа, меньшие M, то чему равна величина максимально возможного потока в любой st-сети с V вершинами и E ребрами? Приведите два ответа — для случаев, когда параллельные ребра разрешены и когда запрещены.

6.4.2. Приведите алгоритм для решения задачи максимального потока для случая, когда сеть представляет собой дерево (без стока).

6.4.3. *Да или нет.* Если да, кратко обоснуйте, почему, а если нет, приведите контрпример:

а) В любом максимальном потоке отсутствует направленный цикл, в котором по каждому ребру проходит положительный поток.

б) Существует максимальный поток, для которого нет направленного цикла, где по каждому ребру проходит положительный поток.

в) Если пропускные способности всех ребер различны, максимальный поток уникален.

г) Если пропускные способности всех ребер увеличить на положительную константу, минимальное сечение не изменится.

д) Если пропускные способности всех ребер умножить на положительное целое число, минимальное сечение не изменится.

27 Зак. 3239

6.4.4. Завершите доказательство утверждения Ж: покажите, что каждый раз, когда какое-то ребро становится критическим, длина расширяющего пути через него должна увеличиться на 2.

6.4.5. Найдите в Интернете большую сеть, которую можно использовать для тестирования потоковых алгоритмов на реальных данных. Возможно, это дорожные сети (автодороги, железные дороги или авиаперевозки), коммуникационные сети (телефонные или компьютерные) или сети распределения товаров. Если пропускные способности недоступны, придумайте разумную модель для их добавления. Напишите программу, которая использует интерфейс для реализации транспортных сетей из ваших данных. При необходимости разработайте дополнительные приватные методы для зачистки данных.

6.4.6. Напишите генератор случайных разреженных сетей с целочисленными пропускными способностями от 0 до 2^{20}. Используйте отдельный класс для пропускных способностей и разработайте две реализации: одна генерирует равномерно распределенные пропускные способности, а другая — нормально распределенные. Реализуйте клиентские программы, которые генерируют случайные сети для обоих распределений весов с указанным множеством значений V и E, чтобы использовать их для эмпирического исследования графов, построенных для различных распределений весов ребер.

6.4.7. Напишите программу, которая генерирует V случайных точек на плоскости, а затем строит транспортную сеть с ребрами, соединяющими (в обоих направлениях) все пары точек, которые находятся в пределах заданного расстояния d друг от друга, и назначает каждому ребру пропускную способность с помощью одной из случайных моделей, описанных в предыдущем упражнении.

6.4.8. *Простые разновидности.* Разработайте клиенты класса `FordFulkerson` для нахождения максимального потока в каждом из следующих видов транспортных сетей:

а) неориентированная;

б) без ограничений на количество источников, или стоков, или ребер, входящих в источник или выходящих из стока;

в) с нижней границей пропускных способностей;

г) с пропускными способностями вершин.

6.4.9. *Распределение товаров.* Пусть поток представляет товары, которые нужно развозить автомобилями между городами, и ребро `u-v` представляет количество товара, которое нужно перевезти из города `u` в город `v` в заданный день. Напишите клиент для вывода ежедневных маршрутных листов водителям, в которых указано, где и сколько товара погрузить и где его выгрузить. Считайте, что ограничения на количество автомобилей нет, и что из заданной точки распределения ничего не вывозится, пока все в нее не завезено.

6.4.10. *Размещение работ.* Разработайте клиент класса `FordFulkerson` для решения задачи размещения работ, используя приведение из утверждения К. Используйте таблицу имен для преобразования символьных имен в целые числа для применения в транспортной сети.

6.4.11. Придумайте семейство задач двудольного сопоставления, где средняя длина расширяющих путей, используемых любым алгоритмом расширения пути для решения соответствующей задачи максимального потока, пропорциональна E.

6.4.12. *st-связность.* Разработайте клиент класса `FordFulkerson`, который для заданного неориентированного графа G и вершин s и t находит минимальное количество ребер в G, удаление которых приведет к отсоединению t от s.

6.4.13. *Непересекающиеся пути.* Разработайте клиент класса `FordFulkerson`, который для заданного неориентированного графа G и вершин s и t находит максимальное количество реберно непересекающихся путей из s в t.

6.5. Сведение и неразрешимость

Сведение

На протяжении этой книги мы формулировали конкретные задачи, а затем разрабатывали алгоритмы и структуры данных для их решения. В некоторых случаях (многие из них перечислены ниже) удобно решать задачу, сформулировав ее как частный случай другой задачи, которую мы уже умеем решать. Формализация этого процесса — полезная отправная точка для изучения взаимосвязей между различными рассмотренными здесь задачами и алгоритмами.

Определение. Говорят, что задача A *сводится* к другой задаче B, если на основе алгоритма решения B можно построить алгоритм решения A.

Похожая концепция применяется в разработке программного обеспечения: при использовании библиотечного метода для решения задачи эта задача сводится к той, которую решает библиотечный метод. В данной книге задачи, которые можно свести к заданной задаче, неформально называются *приложениями*.

Сведения к сортировке

Впервые мы встретились со сведением в главе 2 — для демонстрации того, что эффективный алгоритм сортировки можно применять для эффективного решения многих других задач, которые с виду совсем не связаны с сортировкой. Тогда мы рассмотрели следующие задачи.

- **Нахождение медианы.** Нужно найти значение медианы для заданного множества чисел.
- **Различные значения.** Нужно найти количество различных значений в множестве чисел.
- **Планирование работ для минимизации среднего времени завершения.** Имеется множество работ заданной продолжительности, и нужно запланировать их выполнение на одном процессоре, чтобы минимизировать среднее время завершения.

Утверждение З. К сортировке сводятся следующие задачи.

- Нахождение медианы.
- Подсчет различных значений.
- Планирование работ для минимизации среднего времени выполнения.

Доказательство. См. листинг 2.5.4 и упражнение 2.5.12.

Но при выполнении сведения необходимо учитывать его стоимость. Например, медиану внутри множества чисел можно найти за линейное время, но использование сведения к сортировке приведет к линейно-логарифмическому времени выполнения. Но даже и такие дополнительные затраты могут быть вполне приемлемыми, т.к. при этом используется уже готовая реализация сортировки.

Сортировка ценна по трем причинам.

- Она полезна и сама по себе.
- Имеются эффективные алгоритмы для ее выполнения.
- К ней сводятся многие задачи.

Обычно задачи с такими свойствами называются *моделью решения задачи*. Как и хорошо спроектированные программные библиотеки, хорошо спроектированные модели решения задач могут значительно расширить охват задач, которые мы можем эффективно решать. Одно из опасных мест при рассмотрении моделей решения задач называется *молотком Маслова* — считают, что именно он сформулировал эту максиму в 1960-х годах: *если у вас имеется только молоток, все вокруг кажется гвоздями*. При рассмотрении новых моделей решения задач мы можем использовать их, наподобие молотка Маслова, для решения любой возникающей задачи, а это отвлекает нас от обнаружения лучших алгоритмов для решения этой же задачи, и даже от разработки новых моделей решения задачи. Рассматриваемые нами модели важны, мощны и широко используются, но будет полезно рассматривать и другие возможности.

Сведения к поиску кратчайших путей

В разделе 4.4 мы снова обратились к идее сведения — в контексте алгоритмов поиска кратчайшего пути. В частности, мы рассмотрели следующие задачи.

- **Кратчайшие пути из одного источника в неориентированных графах.** Для заданного *неориентированного* графа со взвешенными ребрами и исходной вершиной s нужно отвечать на запросы вида *Существует ли путь из* s *в заданную вершину* v? Если существует, то нужно найти *кратчайший* такой путь (общий вес которого минимален).
- **Параллельное планирование работ с ограничениями предшествования.** Задано множество работ известной длительности, которые нужно выполнить, и заданы ограничения предшествования, которые указывают, что некоторые работы нужно выполнить до начала выполнения некоторых других работ. Как можно запланировать выполнение работ на идентичных процессорах (их доступно сколько угодно), чтобы все они завершились за минимальное время с учетом ограничений?
- **Арбитраж.** Нужно найти возможность арбитража в заданной таблице курсов обмена валют.

Здесь опять две последние задачи с виду никак не связаны с задачами поиска кратчайших путей, но мы видели, что кратчайшие пути позволяют эффективно решить их. Эти примеры важны, но здесь они приведены просто для иллюстрации. Известно, что к поиску кратчайших путей сводится много важных задач — слишком много, чтобы перечислить здесь их все — т.е. это эффективная и важная модель решения задач.

Утверждение И. К поиску кратчайших путей во взвешенных орграфах сводятся перечисленные ниже задачи.

- Кратчайшие пути из одного источника в неориентированных графах с неотрицательными весами.
- Параллельное планирование с ограничениями предшествования.
- Арбитраж.
- Многие другие задачи.

Примеры в доказательство. См. листинги 4.4.7, 4.4.8 и 4.4.14.

Сведения к нахождению максимального потока

Алгоритмы определения максимального потока также важны в более широком контексте. Из транспортных сетей можно удалять различные ограничения и решать соответствующие задачи, связанные с потоками; можно решать другие задачи на графах и сетях; и можно решать задачи, которые вовсе не связаны с сетями. Вот примеры таких задач.

- **Размещение работ.** Отдел распределения выпускников в колледже устраивает интервью для множества студентов с множеством компаний, в результате чего образуется множество предложений работы (рис. 6.5.1). Если интервью с последующим предложением работы представляет взаимный интерес студента и компании, то для всех будет лучше, если максимизировать количество размещений работ. Можно ли обеспечить работой каждого студента? Каково максимальное количество вакансий, которое можно заполнить?
- **Распределение товара.** У некоторой компании имеются фабрики, где производится один товар; большие склады для временного хранения и распределения; и магазины розничной продажи этого товара. Компании необходимо регулярно развозить товар с фабрик через склады в магазины, используя каналы распределения с различными пропускными способностями. Возможно ли развозить товар со складов в магазины, чтобы удовлетворять другим дополнительным требованиям?
- **Надежность сети.** Компьютерную сеть упрощенно можно рассматривать как состоящую из множества кабелей, соединяющих компьютеры через коммутаторы так, чтобы можно было соединить по кабелям два любых заданных компьютера. Каково минимальное количество кабелей, которые нужно перерезать, чтобы наверняка разъединить некоторую пару компьютеров?

Эти задачи также с виду не связаны одна с другой и с транспортными сетями, но все они сводятся к задаче максимального потока.

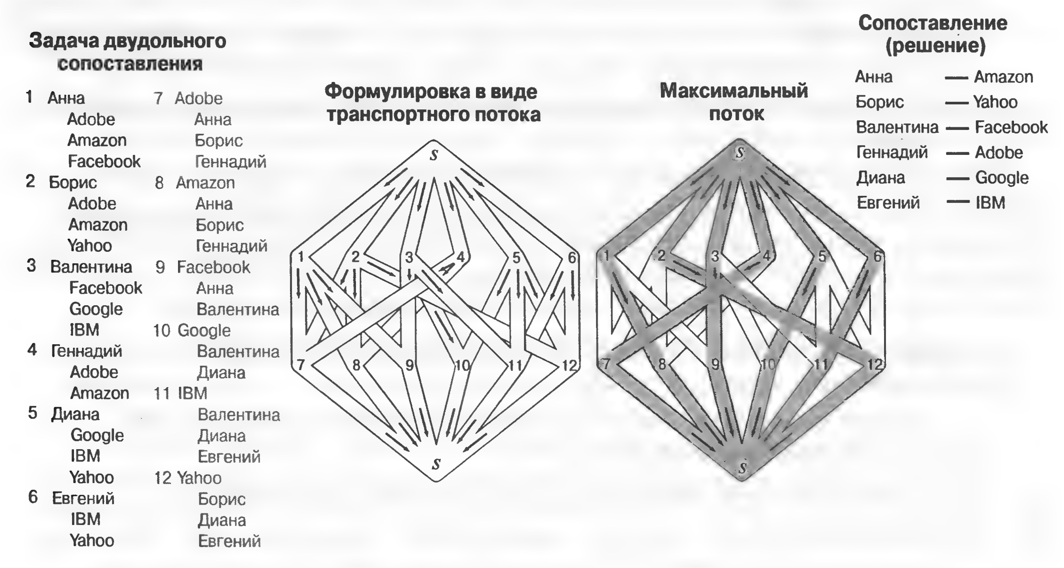


Рис. 6.5.1. Пример сведения задачи максимального двудольного сопоставления к транспортному потоку

Утверждение К. К задаче нахождения максимального потока сводятся следующие задачи.

- Размещение работ.
- Распределение товара.
- Надежность сети.
- Многие другие задачи.

Пример доказательства. Мы докажем возможность первого сведения (которое называется *задачей максимального двудольного сопоставления*), а остальные оставим в качестве самостоятельных упражнений.

Пусть имеется задача размещения работ. Построим экземпляр задачи максимального потока, направив все ребра от студентов к компаниям, добавим вершину-источник с ребрами, ведущими из нее ко всем студентам, и добавим вершину-сток с ребрами, ведущими в нее от всех компаний. Каждому ребру присвоим пропускную способность 1. Любое целочисленное решение задачи максимального потока для этой сети дает решение соответствующей задачи двудольного сопоставления (см. следствие из утверждения Е). Сопоставление соответствует в точности тем ребрам между двумя множествами, которые заполнены алгоритмом нахождения максимального потока до их пропускной способности. Во-первых, транспортный поток всегда дает допустимое сопоставление: поскольку у каждой вершины имеется или входящее (из источника), или исходящее (в сток) ребро пропускной способности 1, то через каждую вершину может пройти не более 1 единицы потока, а это, в свою очередь, означает, что каждая вершина может быть включена только в одно сопоставление. Во-вторых, ни в одном сопоставлении не может быть больше ребер, т.к. каждое такое сопоставление сразу же привело бы к большему потоку, чем найденный алгоритмом максимального потока.

Например, как показано на рис. 6.5.2, алгоритм вычисления максимального потока с помощью расширения пути может использовать пути `s->1->7->t`, `s->2->8->t`, `s->3->9->t`, `s->5->10->t`, `s->6->11->t` и `s->4->7->1->8->2->12->t` и найти сопоставление `1-8`, `2-12`, `3-9`, `4-7`, `5-10` и `6-11`. Значит, в нашем примере существует способ предоставления работы всем студентам. Каждый расширяющий путь заполняет одно ребро из источника и одно ребро в сток. Эти ребра никогда не используются в качестве обратных, поэтому существует не более V расширяющих путей, а общее время вычисления пропорционально VE.

Кратчайшие пути и максимальный поток — важные модели решения задач, т.к. они имеют те же свойства, которые мы сформулировали для сортировки.

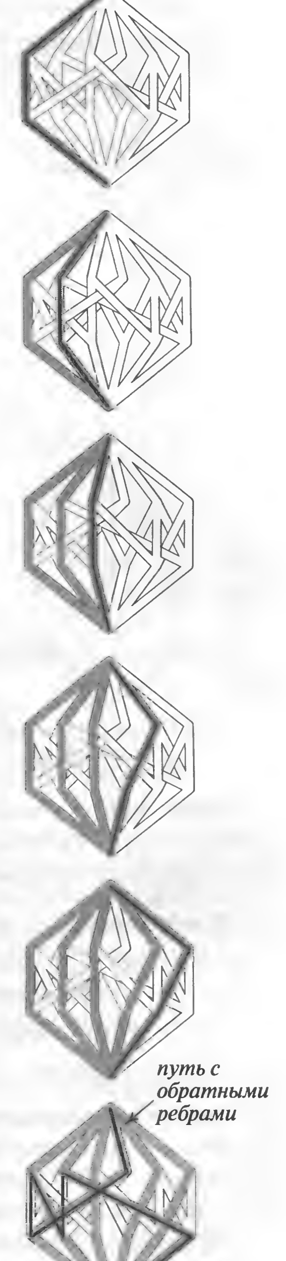


Рис. 6.5.2. Расширяющие пути для двудольного сопоставления

- Они полезны и сами по себе.
- Имеются эффективные алгоритмы для их решения.
- К ним сводятся многие задачи.

Это краткое обсуждение является лишь введением в тему. В курсе исследования операций вы познакомитесь со многими другими задачами, которые сводятся к рассмотренным здесь, и многими другими моделями решения задач.

Требуется максимизировать $f+h$ с учетом ограничений

$$0 \le a \le 2$$
$$0 \le b \le 3$$
$$0 \le c \le 3$$
$$0 \le d \le 1$$
$$0 \le e \le 1$$
$$0 \le f \le 1$$
$$0 \le g \le 2$$
$$0 \le h \le 3$$
$$a = c+d$$
$$b = e+f$$
$$c+e = g$$
$$d+f = h$$

Рис. 6.5.3. Пример задачи ЛП

Линейное программирование

Одним из фундаментальных аспектов исследования операций является *линейное программирование* (ЛП). В нем используется сведение заданной задачи к следующей математической формулировке.

Линейное программирование. Пусть имеется множество M *линейных неравенств* и линейных равенств относительно N переменных и линейная *целевая функция* от N переменных. Нужно найти такие значения переменных, которые максимизируют целевую функцию, или сообщить, что такие значения не существуют (рис. 6.5.3).

Линейное программирование — крайне важная модель решения задач, по следующим причинам.

- Очень много важных задач сводятся к линейному программированию.
- Имеются эффективные алгоритмы для решения задач линейного программирования.

Фраза "полезна сама по себе", которую мы повторяли для других задач, в этом списке не нужна, в силу наличия *огромного количества* практических задач, которые сводятся к линейному программированию.

Утверждение Л. К задаче линейного программирования сводятся перечисленные ниже задачи.

- Максимальный поток.
- Кратчайшие пути.
- Очень многие другие задачи.

Пример доказательства. Мы докажем возможность первого сведения, а второе вынесем в упражнение 6.5.2. Рассмотрим систему неравенств и равенств, в которой каждому ребру соответствует одна переменная и два неравенства, а каждой вершине (кроме источника и стока) — одно равенство. Значением переменной является поток по ребру, неравенства указывают, что данный поток по ребру должен быть в пределах от 0 до пропускной способности ребра, а равенства указывают, что общий поток по ребрам, которые входят в каждую вершину, должен быть равен общему потоку по ребрам, выходящим из этой вершины. Любую задачу максимального потока можно преобразовать таким образом в экземпляр задачи линейного программирования, а решение полученной задачи легко преобразовать в решение задачи максимального потока. На рис. 6.5.4 приведен пример такого преобразования.

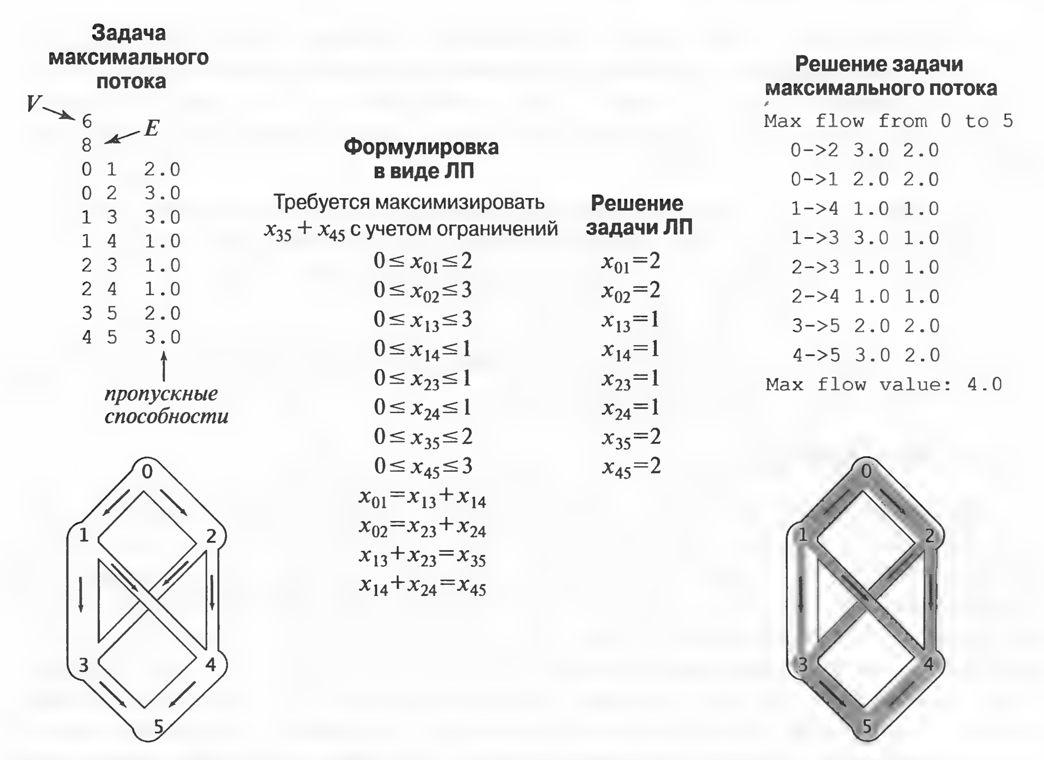


Рис. 6.5.4. *Пример сведения задачи транспортного потока к линейному программированию*

Фраза "очень многие другие задачи" в утверждении Л относится к трем аспектам. Во-первых, *модель очень легко расширить и добавить в нее ограничения*. Во-вторых, *сведение транзитивно*, поэтому все задачи, которые сводятся к нахождению кратчайших путей и максимального потока, также сводятся к линейному программированию. В-третьих, *всевозможные задачи оптимизации можно непосредственно сформулировать в виде задач линейного программирования*. По сути, термин *линейное программирование* означает "формулировка задачи оптимизации в виде задачи линейного программирования", а сам термин *программирование* появился еще до распространения программирования для компьютеров.

Кроме того, что очень многие задачи сводятся к линейному программированию, важно и то, что уже много десятилетий известных эффективные алгоритмы для решения этой задачи. Наиболее известен из них *симплекс-алгоритм*, разработанный Г. Дантцигом (G. Dantzig) в 1940-х годах. Его нетрудно понять (см. простейшую реализацию на сайте книги). Позднее Л.Г. Хачян в 1979 г. предложил *алгоритм эллипсоидов*, который привел к разработке *методов внутренней точки* в 1980-х годах и зарекомендовал себя эффективным дополнением симплекс-метода для очень больших задач линейного программирования, которые возникают в современных приложениях. В настоящее время для решения задач линейного программирования имеются надежные, тщательно проверенные и эффективные инструменты, на которых основывается работа современных корпораций. Кроме того, постоянно ширится их применение в научных областях и даже в прикладном программировании. Если вы можете смоделировать задачу в виде задачи линейного программирования, то вы, скорее всего, сможете решить ее.

Без преувеличения можно сказать, что линейное программирование является прародителем моделей решения задач, поскольку очень много задач сводится к нему. И естественно возникает вопрос: существует ли модель решения задач, еще более мощная, чем линейное программирование? Какие виды задач *не* сводятся к линейному программированию? Вот пример такой задачи.

> ***Балансировка нагрузки.*** Имеется множество работ определенной длительности, которые нужно выполнить. Как можно запланировать выполнение этих работ на двух идентичных процессорах, чтобы минимизировать время завершения всех задач?

Можно ли сформулировать более общую модель решения задач и эффективно решать экземпляры этих задач в пределах такой модели? Этот вопрос приводит нас к нашей последней теме — *неразрешимости*.

Неразрешимость

Алгоритмы, описанные в данной книге, обычно применяются для решения практических задач и поэтому потребляют значительные объемы ресурсов. Практическая польза большинства алгоритмов очевидна, и для многих задач у нас есть даже выбор из нескольких эффективных вариантов. К сожалению, на практике возникают и многие другие задачи, для которых не найдены такие эффективные решения. И что еще хуже, для большого класса таких задач невозможно даже сказать, существует или нет эффективное решение. Это очень не нравилось программистам и конструкторам алгоритмов, которые не могли отыскать эффективные алгоритмы для решения многих практических задач, а также для теоретиков, которые не могли найти доказательства, что эти задачи действительно трудны. В этой области были проведены интенсивные исследования, приведшие к разработке механизмов, которые позволяют классифицировать новые задачи как "трудноразрешимые" в конкретном техническом смысле. Значительная часть этой работы выходит за рамки данной книги, но основные идеи понять совсем не трудно. Мы изложим их здесь потому, что каждый программист при встрече с новой задачей должен иметь некоторое представление, что существуют задачи, для которых неизвестны гарантированно эффективные алгоритмы.

Фундамент. Одним из наиболее красивых и интересных интеллектуальных открытий XX века была *машина Тьюринга*, придуманная А. Тьюрингом (A. Turing) в 1930-х годах. Это простая модель вычислений, но она достаточно общая, чтобы охватить любую компьютерную программу или вычислительное устройство. Машина Тьюринга представляет собой автомат с конечным числом состояний, который может читать входные данные, переходить из состояния в состояние и выводить результаты. И она сформировала основание теории вычислений, начиная со следующих двух принципов.

- *Универсальность.* Все физически реализуемые вычислительные устройства можно моделировать с помощью машины Тьюринга. Этот принцип называется *тезисом Черча-Тьюринга*, он касается реального мира, и поэтому его невозможно доказать (хотя можно опровергнуть). Свидетельством в пользу этого тезиса является то, что математики и специалисты по вычислениям разработали многочисленные вычислительные модели, и доказано, что все они эквивалентны машине Тьюринга.
- *Вычислимость.* Существуют задачи, которые невозможно решить с помощью машины Тьюринга (или, согласно принципу универсальности, на любом другом вычислительном устройстве). Это математически доказанный факт. Известным

примером такой задачи является *задача остановки*: ни одна программа не может гарантированно определить, остановится ли когда-либо некоторая заданная программа.

В нашем случае нас интересует третий принцип, который относится к эффективности вычислительных устройств.

- *Расширенный тезис Черча-Тьюринга*. Порядок роста времени выполнения программы, которая решает задачу на любом вычислительном устройстве, не превышает полинома от порядка роста времени для некоторой программы, которая решает эту задачу на машине Тьюринга (или любом другом вычислительном устройстве).

Этот принцип также относится к реальному миру, но он дополнен фактом, что все известные вычислительные устройства можно моделировать с помощью машины Тьюринга, с не более чем полиномиальным увеличением стоимости. В последнее время принцип *квантовых вычислений* дал некоторым исследователям повод для сомнений в тезисе Черча-Тьюринга. Большинство продолжают считать, что с практической точки зрения он некоторое время еще будет достаточно надежен, но многие уже стараются найти доказательство его ложности.

Экспоненциальное время выполнения

Теория неразрешимости старается отделить задачи, которые можно решить за полиномиальное время, от задач, которые (видимо) требуют *экспоненциального* времени для решения в худшем случае. Алгоритм с экспоненциальным временем выполнения полезно рассматривать как алгоритм, который для входных данных размера N требует времени, пропорционального (не менее чем) 2^N. Суть алгоритма не изменится, если заменить 2 любым другим числом $\alpha > 1$. Обычно считается, что алгоритм с экспоненциальным временем работы не может гарантированно решить задачу размером (скажем) 100 за приемлемое время: ведь никто не сможет ждать, пока алгоритм выполнит 2^{100} шагов, независимо от скорости компьютера. Экспоненциальный рост сводит на нет все технологические достижения: какой-нибудь суперкомпьютер может работать в триллион раз быстрее старинного абака, но ни один из них не может подобраться к решению задачи, которая требует выполнения 2^{100} шагов. Иногда границу между "легко" и "трудно" нелегко выявить. Например, в разделе 4.1 был рассмотрен алгоритм, который может решить такую задачу.

Длина кратчайшего пути. Какова длина кратчайшего пути из заданной вершины s в заданную вершину t в заданном графе?

Но мы не рассматривали алгоритм для решения следующей, с виду почти такой же, задачи.

Длина самого длинного пути. Какова длина *самого длинного* простого пути из заданной вершины s в заданную вершину t в заданном графе?

Однако, насколько нам известно, эти задачи находятся почти на противоположных концах спектра сложности. Поиск в ширину дает решение первой задачи за *линейное* время, но все известные алгоритмы для решения второй задачи требуют в худшем случае *экспоненциального* времени. Код в листинге 6.5.1 содержит вариант поиска в глубину, который решает такую задачу. Он очень похож на обычный поиск в глубину, но просматривает *все* простые пути из s в t в орграфе, чтобы выбрать самый длинный из них.

Листинг 6.5.1. Нахождение длины самого длинного пути в графе

```
public class LongestPath
{
   private boolean[] marked;
   private int max;

   public LongestPath(Graph G, int s, int t)
   {
      marked = new boolean[G.V()];
      dfs(G, s, t, 0);
   }

   private void dfs(Graph G, int v, int t, int i)
   {
      if (v == t && i > max) max = i;
      if (v == t) return;
      marked[v] = true;
      for (int w : G.adj(v))
         if (!marked[w]) dfs(G, w, t, i+1);
      marked[v] = false;
   }

   public int maxLength()
   {  return max;  }
}
```

Задачи поиска

Огромный разрыв между задачами, которые можно решить “эффективными” алгоритмами вроде тех, которые были рассмотрены в данной книге, и задачами, где решение нужно искать среди фактически необъятного количества возможных вариантов, позволяет изучать интерфейс между ними с помощью простой формальной модели. Сначала нужно охарактеризовать вид изучаемой задачи.

> **Определение.** *Задача поиска* — это задача со следующим свойством: время, необходимое, чтобы *удостовериться*, что любое ее решение корректно, ограничено полиномом от размера входных данных. Говорят, что алгоритм *решает* задачу поиска, если для любых входных данных он или выдает решение, или сообщает, что решение не существует.

Вот четыре конкретные задачи, которые мы рассмотрим при обсуждении неразрешимости.

- **Выполнимость линейных уравнений.** Для заданной системы *M линейных уравнений* с *N* переменными нужно найти такие значения переменных, которые удовлетворяют всем равенствам, или сообщить, что это невозможно.
- **Выполнимость линейных неравенств (задача линейного программирования в виде задачи поиска).** Для заданной системы *M линейных неравенств* с *N* переменными нужно найти такие значения переменных, которые удовлетворяют всем неравенствам, или сообщить, что это невозможно.
- **Выполнимость линейных неравенств в целых числах 0–1 (задача линейного программирования в целых числах 0–1 в виде задачи поиска).** Для заданной системы *M линейных неравенств* с *N целочисленными* переменными нужно найти такие значе-

ния переменных, равные 0 или 1, которые удовлетворяют всем неравенствам, или сообщить, что это невозможно.

- **Логическая выполнимость.** Для заданной системы *M уравнений* с *N логическими* переменными, содержащих операции *и* и *или*, нужно найти такие значения переменных, которые удовлетворяют всем равенствам, или сообщить, что это невозможно.

Эти задачи называются задачами *выполнимости*. И для установления, что некоторая задача является задачей поиска, достаточно показать, что любое ее решение достаточно хорошо охарактеризовано, чтобы можно было эффективно подтвердить его корректность. Решение задачи поиска похоже на поиск иголки в стоге сена, когда про иголку известно только то, что вы сможете узнать ее, когда увидите. Например, для какого-то заданного набора значений переменных в любой из вышеперечисленных задач выполнимости можно легко проверить, выполняется или нет каждое равенство или неравенство, но поиск такого набора значений — совсем другая задача. Для обозначения таких задач часто используется термин NP (его смысл будет объяснен чуть ниже).

Определение. NP — это множество всех задач поиска.

NP — всего лишь точная характеристика всех задач, которые ученые, инженеры и программисты *надеются решить* с помощью программ, выполняющихся за приемлемое время.

Другие виды задач

Концепция задач поиска — один из многих способов охарактеризовать множество задач, которые формируют основу для изучения неразрешимости. Другие возможные концепции — задачи *принятия решений* (существует ли решение?) и задачи *оптимизации* (каково наилучшее решение?) Например, приведенная выше задача поиска самого длинного пути является задачей оптимизации, а не поиска (при наличии решения невозможно проверить, что это максимальная длина пути). Поисковый вариант этой задачи — нужно *найти* простой путь, соединяющий все вершины (так называемая *задача гамильтонова пути*). А вариант с принятием решения — вопрос, *существует ли* простой путь, соединяющий все вершины. Арбитраж, логическая выполнимость и гамильтонов путь являются задачами поиска; вопрос, существует ли какое-либо решение любой из этих задач, является задачей принятия решения; а самый короткий или длинный путь, максимальный поток и линейное программирование представляют собой задачи оптимизации. Задачи поиска, принятия решений и оптимизации технически не эквивалентны, но обычно их можно свести друг к другу (см. упражнения 6.58 и 6.59), а основные выводы применимы ко всем трем типам задач.

Легкие задачи поиска

В определении NP (в совокупности с предыдущим определением) ничего не говорится о сложности *нахождения* решения — просто о проверке, что это действительно решение. Второе из двух множеств задач, которые лежат в основе изучения неразрешимости — оно называется P — связано со сложностью нахождения решения. В этой модели эффективность алгоритма представляет собой функцию от количества битов, которыми закодированы входные данные.

Определение. P — это множество всех задач поиска, которые можно решить за полиномиальное время.

В этом определении неявно подразумевается, что полиномиальная граница — это граница *в худшем случае*. Чтобы задача принадлежала множеству P, должен существовать алгоритм, который *гарантированно* решает ее за полиномиальное время. Сам полином никак не оговаривается. Линейное время, линейно-логарифмическое, квадратичное, кубичное — все это полиномиальные временные границы, и под наше определение попадают все изученные нами в этой книге стандартные алгоритмы. Время работы алгоритма зависит от используемого компьютера, однако расширенный тезис Черча-Тьюринга утверждает, что существование решения с полиномиальным временем на любом вычислительном устройстве означает существование решения с полиномиальным временем на любом другом вычислительном устройстве.

Сортировка принадлежит множеству P, т.к. (к примеру) сортировка вставками выполняется за время, пропорциональное N^2 (существование линейно-логарифмических алгоритмов сортировки сейчас не важно). То же относится и к поиску кратчайших путей, выполнимости линейных уравнений и многим другим задачам. Наличие эффективного алгоритма для решения задачи является доказательством, что задача принадлежит P. То есть множество P — просто точное описание всех задач, которые ученые, инженеры и программисты *решают* с помощью программ, гарантированно отрабатывающих за приемлемое время.

Недетерминизм

Буква N в обозначении NP означает *недетерминизм* (неповторяемость). Это принцип, согласно которому (теоретически) можно увеличить мощь компьютеров, снабдив их свойством недетерминизма: когда алгоритму необходимо выбрать один из нескольких вариантов, он может "угадать" правильный. В контексте нашего обсуждения можно считать, что алгоритм для недетерминистской машины "угадывает" решение задачи, а мотом проверяет правильность этого решения. В машине Тьюринга недетерминизм означает просто определение для заданного состояния и входных данных двух различных последующих состояний, как допустимых путей к искомому результату. Недетерминизм может быть математической фантазией, но это полезная фантазия. Например, в разделе 5.4 недетерминизм был использован в качестве средства для построения алгоритма: наш алгоритм сопоставления с регулярными выражениями основан на эффективном моделировании недетерминистской машины. В табл. 6.5.1 приведены примеры NP-задач, а в табл. 6.5.2 — примеры задач из множества P.

Таблица 6.5.1. Примеры NP-задач

Задача	Входные данные	Описание	Полином. время	Пример	Решение
Гамильтонов путь	Граф G	Найти простой путь, проходящий через каждую вершину	?		0-2-1-3
Разложение на множители	Целое число x	Найти нетривиальный делитель числа x	?	97605257271	8784561
Выполнимость линейных неравенств в целых числах 0–1	M переменных 0–1 N неравенств	Присвоить переменным значения, удовлетворяющие неравенствам	?	$x - y \leq 1$ $2x - z \leq 2$ $x + y \geq 2$ $z \geq 0$	$x = 1$ $y = 1$ $z = 0$
Все задачи		См. табл. 6.5.2			

Таблица 6.5.2. Примеры задач из множества P

Задача	Входные данные	Описание	Алгоритм с полином. временем	Пример	Решение
Кратчайший st-путь	Граф G вершины s, t	Найти кратчайший путь из s в t	Поиск в ширину		0-3
Сортировка	Массив a	Найти перестановку, упорядочивающую элементы a	Сортировка слиянием	2 1 0 3	0 1 2 3
Выполнимость линейных уравнений	M переменных N уравнений	Присвоить переменным значения, удовлетворяющие равенствам	Гауссово исключение	$x + y = 1,5$ $2x - y = 0$	$x = 0,5$ $y = 1$
Выполнимость линейных неравенств	M переменных N неравенств	Присвоить переменным значения, удовлетворяющие неравенствам	Метод эллипсоидов	$x - y \leq 1,5$ $2x - z \leq 0$ $x + y \geq 3,52$ $z \geq 4,0$	$x = 2,0$ $y = 1,5$ $z = 4,0$

Основной вопрос

Недетерминизм — настолько мощное понятие, что его как-то несерьезно рассматривать вообще. Зачем тратить силы на рассмотрение воображаемого средства, которое превращает сложные задачи в тривиальные? А ответ в том, что несмотря на всю (казалось бы) мощь недетерминизма, никто не смог *доказать*, что он помогает решить хоть какую-то конкретную задачу! Точнее, никто не смог найти ни одну задачу, для которой можно доказать, что она принадлежит NP, но не P (или даже доказать, что такая задача существует), что оставляет открытым следующий вопрос.

Совпадает ли P с NP?

Этот вопрос впервые был поставлен в знаменитом письме К. Геделя к Дж. фон Нейману в 1950 г. и с тех пор вводит в ступор математиков и компьютерных теоретиков. Другие способы постановки этого вопроса более явно демонстрируют его фундаментальную природу.

- Существуют ли *хоть какие-то* трудноразрешимые задачи поиска?
- Можно ли будет решать некоторые задачи более эффективно, если мы сможем создать недетерминистское вычислительное устройство?

Отсутствие ответов на эти вопросы весьма удручает, т.к. многие важные практические задачи принадлежат NP, но могут принадлежать, а могут и не принадлежать P (лучшие известные детерминистские алгоритмы могут выполняться за экспоненциальное время). Если мы сможем доказать, что некоторая задача не принадлежит P, то мы сможем прекратить поиск эффективного ее решения. А при отсутствии такого доказательства существует вероятность, что где-то затаился эффективный алгоритм. Вообще-то, с учетом современного уровня наших знаний может существовать эффективный алгоритм для *каждой* задачи в NP — из чего следует, что многие эффективные алгоритмы еще ждут своего открытия. Почти никто не верит, что P = NP, и существенные усилия были

потрачены на доказательство обратного, но до сих пор это открытая исследовательская задача в компьютерных науках.

Сведения за полиномиальное время

В начале этого раздела было показано, что если задача A *сводится* к другой задаче B, то нужно выполнить три шага.

- Преобразование задачи A в экземпляр задачи B.
- Решение этого экземпляра B.
- Преобразование решения задачи B в решение задачи A.

Задача логической выполнимости

$$(x_1' \text{ or } x_2 \text{ or } x_3) \text{ and}$$
$$(x_1 \text{ or } x_2' \text{ or } x_3) \text{ and}$$
$$(x_1' \text{ or } x_2' \text{ or } x_3') \text{ and}$$
$$(x_1' \text{ or } x_2' \text{ or } x_3)$$

Формулировка задачи выполнимости системы линейных неравенств с целыми переменными 0–1

c_1 равно 1 тогда и только тогда, когда первое высказывание выполнимо

$$c_1 \geq 1 - x_1$$
$$c_1 \geq x_2$$
$$c_1 \geq x_3$$
$$c_1 \leq (1 - x_1) + x_2 + x_3$$

$$c_2 \geq x_1$$
$$c_2 \geq 1 - x_2$$
$$c_2 \geq x_3$$
$$c_2 \leq x_1 + (1 - x_2) + x_3$$

$$c_3 \geq 1 - x_1$$
$$c_3 \geq 1 - x_2$$
$$c_3 \geq 1 - x_3$$
$$c_3 \leq (1 - x_1) + 1 - x_2 + (1 - x_3)$$

$$c_4 \geq 1 - x_1$$
$$c_4 \geq 1 - x_2$$
$$c_4 \geq x_3$$
$$c_4 \leq (1 - x_1) + (1 - x_2) + x_3$$

s равно 1 тогда и только тогда, когда все c равны 1

$$s \leq c_1$$
$$s \leq c_2$$
$$s \leq c_3$$
$$s \leq c_4$$
$$s \geq c_1 + c_2 + c_3 + c_4 - 3$$

Рис. 6.5.5. Пример сведения задачи логической выполнимости к выполнимости линейных неравенств с целыми переменными 0–1

Если мы можем эффективно выполнить преобразования и решить задачу B, то мы можем эффективно решить задачу A. В данном контексте под словом “эффективно” имеется в виду, что *A сводится к B за полиномиальное время*. Раньше мы использовали сведение, чтобы продемонстрировать пользу моделей решения задач, которые могут существенно расширить спектр задач, разрешимых с помощью эффективных алгоритмов. Теперь же мы используем сведение в другом смысле — *для доказательства трудноразрешимости задачи*. Если известно, что задача A трудноразрешима, и A сводится за полиномиальное время к B, то и B должна быть тоже трудноразрешимой. Иначе гарантированное полиномиальное решение B даст гарантированное полиномиальное решение A.

Утверждение М. Задача логической выполнимости сводится за полиномиальное время к задаче выполнимости системы линейных неравенств с целочисленными значениями 0–1.

Доказательство. Пусть имеется экземпляр задачи логической выполнимости. Определим систему неравенств с переменными 0–1, каждая из которых соответствует одной логической переменной, и по одной переменной 0–1, соответствующей каждому утверждению, как показано на рис. 6.5.5. Это построение позволяет преобразовать решение задачи выполнимости линейных неравенств с целочисленными значениями в решение задачи логической выполнимости, присвоив каждой логической переменной значение *true*, если соответствующая целочисленная переменная равна 1, и *false*, если она равна 0.

Следствие. Если задача выполнимости трудноразрешима, то таковой является и задача целочисленного линейного программирования.

Это существенное заявление об относительной сложности решения данных двух задач даже при отсутствии точного определения *трудноразрешимости*. В данном контексте этот термин означает "не принадлежит P". Но обычно для обозначения задач, не принадлежащих P, используется слово *неразрешимая*. Начиная с пионерской работы Р. Карпа в 1972 г., исследователи показали, что буквально десятки тысяч задач из очень широкого спектра прикладных областей связаны между собой подобными отношениями. Более того, эти отношения означают гораздо больше, чем просто взаимосвязь двух отдельных задач — и об этом мы сейчас поговорим.

NP-полнота

Про очень многие задачи известно, что они принадлежат NP, но, возможно, не принадлежат P. То есть нетрудно *проверить* правильность любого заданного решения, но, несмотря на значительные усилия, никто не смог разработать алгоритм, который позволяет *найти* какое-то решение. Все эти многочисленные задачи обладают одним дополнительным свойством, которое также свидетельствует в пользу утверждения $P \neq NP$:

Определение. Задача поиска *A* называется *NP-полной*, если все задачи в NP сводятся к *A* за полиномиальное время.

Это определение позволяет уточнить наше определение трудноразрешимости: оно означает "неразрешимо, если P = NP". Если *любую* NP-полную задачу можно решить за полиномиальное время на детерминистской машине, то это же верно и для *всех задач из NP* (т.е. P = NP). Другими словами, коллективную неспособность всех исследователей найти эффективные алгоритмы для всех этих задач можно рассматривать как коллективную неспособность доказать, что P = NP. Для NP-полных задач мы не ожидаем найти алгоритмы с гарантированно полиномиальным временем выполнения. Про большинство практических задач поиска известно, что они либо принадлежат P, либо NP-полны.

Теорема Кука–Левина

Сведение использует NP-полноту одной задачи, чтобы вывести NP-полноту другой. Но сведение нельзя использовать в одном случае: как доказать NP-полноту для *первой* задачи? Это было сделано независимо С. Куком (S. Cook) и Л. Левиным (L. Levin) в начале 1970-х годов.

Утверждение H (теорема Кука-Левина). Задача логической выполнимости NP-полна.

Очень краткий набросок доказательства. Нам нужно показать, что если существует алгоритм для задачи логической выполнимости с полиномиальным временем выполнения, то все задачи в NP можно решить за полиномиальное время. Недетерминистская машина Тьюринга может решить любую задачу из NP, поэтому вначале нужно выразить все характеристики этой машины в логических формулах, наподобие тех, которые появляются в задаче логической выполнимости. Это построение устанавливает соответствие между каждой задачей в NP (которую можно записать в виде программы для недетерминистской машины Тьюринга) и некоторым экземпляром задачи выполнимости (перевод этой программы на язык логических формул). А решение задачи выполнимости, по сути, соответствует моделированию машины, на которой выполняется заданная программа с заданными входными данными, поэтому оно порождает решение для экземпляра данной задачи. Дальнейшее изложение доказательства далеко выходит за рамки данной книги. К счастью, в реальности достаточно одного такого доказательства: для установления NP-полноты гораздо легче использовать сведение.

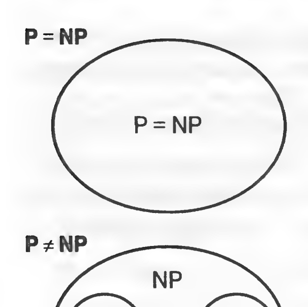


Рис. 6.5.6. Две возможные вселенные

Теорема Кука-Левина, в сочетании со многими тысячами сведений с полиномиальным временем от NP-полных задач, оставляет нас в одной из двух возможных вселенных (рис. 6.5.6): либо P = NP, и неразрешимые задачи поиска не существуют (все задачи поиска можно решить за полиномиальное время); либо P ≠ NP, и существуют неразрешимые задачи поиска (некоторые задачи поиска невозможно решить за полиномиальное время).

NP-полные задачи часто возникают в важных практических приложениях, и поэтому естественно стремление найти хорошие алгоритмы для их решения. То, что ни для одной из этих задач не найден хороший алгоритм, несомненно, является серьезным свидетельством в пользу того, что P ≠ NP, и большинство исследователей уверены, что так оно и есть. Однако никто и не может доказать, что любая из этих задач не принадлежит P, и это также серьезное свидетельство в пользу другой вселенной. Но верно ли, что P = NP, или нет — практически важно то, что наилучшие известные алгоритмы для любых NP-полных задач требуют в худшем случае экспоненциального времени выполнения.

Классификация задач

Чтобы доказать, что некая задача поиска принадлежит P, необходимо предъявить алгоритм ее решения за полиномиальное время — возможно, с помощью сведения ее к задаче, о которой известно, что она принадлежит P. Чтобы доказать, что задача из NP является NP-полной, необходимо показать, что к ней можно свести какую-то известную задачу за полиномиальное время: т.е. что для решения новой NP-полной задачи можно использовать алгоритм с полиномиальным временем выполнения, а, следовательно, и для решения всех задач из NP. Таким образом была показана NP-полнота многих тысяч задач, как это было сделано для целочисленного линейного программирования в утверждении M. Список в табл. 6.5.3 содержит некоторые задачи, рассмотренные Карпом; это вполне представительный список, но он содержит лишь мизерную долю известных NP-полных задач. Отнесение задачи к легкоразрешимым (принадлежит P) или трудноразрешимым (NP-полные) может обладать одной из следующих характеристик.

- *Очевидное.* Например, широко известный алгоритм исключения Гаусса доказывает, что задача выполнимости системы линейных уравнений принадлежит P.
- *Хитроумное, но не сложное.* К примеру, разработка доказательства наподобие доказательства утверждения M требует некоторого опыта и практики, но его нетрудно понять.
- *Крайне сложное.* Например, задачу линейного программирования долго не удавалось классифицировать, но алгоритм эллипсоидов Хачяна доказывает, что линейное программирование принадлежит P.
- *Открытое.* К примеру, задача *изоморфизма графов* (для заданных двух графов нужно найти способ такого переименования вершин, чтобы один из них стал идентичен другому) и задача *факторизации* (для заданного натурального числа нужно найти нетривиальный делитель) до сих пор не классифицированы.

Это плодородная и активная область современных исследований, в которой каждый год появляются тысячи новых статей. Как видно из нескольких последних элементов табл. 6.5.3, при этом затронуты все научные области. Вспомните, что наше определение NP охватывает задачи, которые ученые, инженеры и программисты *надеются решить* приемлемым способом — конечно, все такие задачи необходимо классифицировать!

Таблица 6.5.3. Некоторые знаменитые NP-полные задачи

Задача	Формулировка
Логическая выполнимость	Задана система *M* уравнений с *N* логическими переменными. Нужно найти значения переменных, которые удовлетворяют всем уравнениям, или сообщить, что такие значения не существуют
Целочисленное линейное программирование	Задана система *M* линейных неравенств с *N* целочисленными переменными. Нужно найти значения переменных, которые удовлетворяют всем неравенствам, или сообщить, что такие значения не существуют
Балансировка нагрузки	Имеется множество заданий известной длительности, которые нужно выполнить, и временная граница *T*. Нужно составить график выполнения этих заданий на двух идентичных процессорах, чтобы выполнить их все до наступления времени *T*
Вершинное покрытие	Задан граф и целое значение *C*. Нужно найти множество *C* вершин, таких, что каждое ребро графа инцидентно по крайней мере одной вершине из этого множества
Гамильтонов путь	Задан граф, и нужно найти простой путь, который проходит через каждую вершину точно один раз, или сообщить, что такой путь не существует
Свертывание белка	Задан уровень энергии *M*, и нужно найти свернутую трехмерную структуру белка с потенциальной энергией, меньшей *M*
Модель Айзинга	Задана модель Айзинга трехмерной кристаллической решетки и предельное значение энергии *E*. Нужно определить, существует ли подграф со свободной энергией, меньшей *E*
Портфель рисков для заданной отдачи	Задан портфель инвестиций с заданной общей стоимостью, требуемая отдача, значения рисков для каждой инвестиции и пороговое значение *M*. Нужно найти способ такого распределения инвестиций, чтобы риск был меньше *M*

Как справиться с NP-полнотой

Для каждой из этого огромного множества задач нужно хоть какое-то решение, поэтому естественен интерес в нахождении способов их решения. Все такие способы невозможно охватить в одном параграфе, но можно кратко описать различные подходы, которые уже опробованы. Один из подходов — изменить задачу и найти "приблизительный" алгоритм, который находит не лучшее решение, но гарантированно близкое к лучшему. Например, легко найти решение задачи коммивояжера в евклидовом пространстве, которое не дотягивает до оптимального не более чем в 2 раза. К сожалению, при поиске улучшенных приближений этот подход не всегда может отодвинуть NP-полноту. Другой подход — разработка алгоритма, эффективно решающего практически все экземпляры задачи, которые могут возникнуть на практике, хотя существуют входные

данные, для которых поиск решения слишком сложен. Наиболее известным примером такого подхода являются методы решения для целочисленного линейного программирования — "рабочие лошадки" в области оптимизационных задач огромного размера, которые используются уже много десятилетий в различных промышленных приложениях. В принципе они могут потребовать для работы экспоненциального времени, но входные данные, которые возникают на практике, не представляют собой такой худший случай. Третий подход — работать с "эффективными" экспоненциальными алгоритмами, дополненными техникой *отката*, которая позволяет не проверять все возможные решения. И, наконец, существует немалый промежуток между полиномиальным и экспоненциальным временем, о котором теория умалчивает. Как вам алгоритм, который выполняется за время, пропорциональное $N^{\log N}$ или $2^{\sqrt{N}}$?

Все прикладные области, рассмотренные в данной книге, имеют отношение к NP-полноте: NP-полные задачи возникают в элементарном программировании, сортировке и поиске, обработке графов, обработке строк, научном программировании, системном программировании, исследовании операций и во всех других областях, где требуются вычисления. Наиболее важный практический аспект теории NP-полноты — механизм определения, является ли новая задача из любой из таких областей "легкой" или "трудной". Если кто-то нашел эффективный алгоритм для решения новой задачи, то все отлично. Но если нет, доказательство NP-полноты задачи показывает, что разработка эффективного алгоритма будет невероятным достижением (и что, видимо, следует поискать какой-то другой подход). Множество эффективных алгоритмов, которые мы изучили в данной книге, свидетельствует, что в области эффективных вычислительных методов мы существенно продвинулись вперед со времен Евклида, но теория NP-полноты показывает, что нам еще предстоит узнать гораздо больше.

Упражнения

6.5.1. Найдите нетривиальный делитель числа 37703491.

6.5.2. Докажите, что задача поиска кратчайших путей сводится к линейному программированию.

6.5.3. Возможен ли алгоритм, который решает NP-полную задачу за среднее время $N^{\log N}$, если $P \neq NP$? Обоснуйте ответ.

6.5.4. Допустим, кто-то открыл алгоритм, который гарантированно решает задачу логической выполнимости за время, пропорциональное $1{,}1^N$. Означает ли это, что и другие NP-полные задачи можно решить за время, пропорциональное $1{,}1^N$?

6.5.5. Насколько значительной будет программа, которая сможет решать задачу целочисленного линейного программирования за время, пропорциональное $1{,}1^N$?

6.5.6. Приведите сведение за полиномиальное время задачи вершинного покрытия к задаче выполнимости линейных неравенств с целочисленными переменными 0–1.

6.5.7. Докажите, что задача поиска гамильтонова пути в *ориентированном* графе является NP-полной, воспользовавшись фактом NP-полноты задачи поиска гамильтонова пути для неориентированных графов.

6.5.8. Пусть известно, что две задачи являются NP-полными. Означает ли это, что существует сведение одной из них к другой за полиномиальное время?

6.5.9. Пусть X — NP-полная задача, сводимая к задаче Y за полиномиальное время, а Y сводится за полиномиальное время к X. Обязательно ли Y является NP-полной?

Ответ: нет, Y может не принадлежать NP.

6.5.10. Пусть имеется алгоритм для решения варианта задачи логической выполнимости в виде задачи принятия решений. Это означает, что существует и присваивание переменным логических значений, которые удовлетворяют логическому выражению. Покажите, как можно найти такой набор значений.

6.5.11. Пусть имеется алгоритм для решения варианта задачи вершинного покрытия в виде задачи принятия решений. Это означает, что существует вершинное покрытие заданного размера. Покажите, как можно решить такой оптимизационный вариант задачи поиска покрытия минимальным множеством вершин.

6.5.12. Объясните, почему оптимизационный вариант задачи вершинного покрытия не обязательно является задачей поиска.

6.5.13. Пусть X и Y — две задачи поиска, и X сводится к Y за полиномиальное время. Какие из следующих утверждений верны?

а) Если NP-полна Y, то NP-полна и X.

б) Если NP-полна X, то NP-полна и Y.

в) Если X принадлежит P, то и Y принадлежит P.

г) Если Y принадлежит P, то и X принадлежит P.

6.5.14. Допустим, что P $\neq$ NP. Какие из следующих утверждений верны?

а) Если задача NP-полна, то ее невозможно решить за полиномиальное время.

б) Если задача принадлежит NP, то ее невозможно решить за полиномиальное время.

в) Если задача принадлежит NP, но не является NP-полной, то ее можно решить за полиномиальное время.

г) Если задача принадлежит P, то она не является NP-полной.

Предметный указатель

И

К

Л

М

Н

О

П

Р

Алгоритмы на Java

4-Е ИЗДАНИЕ



Самая важная информация об алгоритмах и структурах данных

КЛАССИЧЕСКИЙ СПРАВОЧНИК
Последнее издание из серии бестселлеров Седжвика, содержащее самый важный объем знаний, наработанных за последние несколько десятилетий.

ШИРОКИЙ СПЕКТР РАССМАТРИВАЕМЫХ ТЕМ
Полное описание структур данных и алгоритмов для сортировки, поиска, обработки графов и строк, включая пятьдесят алгоритмов, которые должен знать каждый программист.
См. http://algs4.cs.princeton.edu/code.

ПОЛНОСТЬЮ ПЕРЕРАБОТАННЫЙ КОД
Новые реализации, написанные на языке Java, в доступном стиле модульного программирования — весь код предоставлен пользователю и готов к применению.

ПРИВЯЗКА К ПРАКТИКЕ
Алгоритмы изучаются в контексте важных научных, технических и коммерческих приложений. Клиентские программы и алгоритмы записаны в реальном коде, а не на псевдокоде, как во многих других книгах.

НАУЧНЫЙ ПОДХОД
Вывод точных оценок производительности на основе соответствующих математических моделей и эмпирических тестов, подтверждающих эти модели.

ДОСТУПНОСТЬ МАТЕРИАЛОВ НА ВЕБ-САЙТЕ
На свободно доступном и исчерпывающем веб-сайте http://algs4.cs.princeton.edu находятся дайджесты, коды программ, тестовые данные, программные проекты, упражнения, слайды для лекций и другие полезные ресурсы.

СОДЕРЖАНИЕ

Категория: программирование на Java
Предмет рассмотрения: алгоритмы
Уровень: для пользователей средней и высокой квалификации



www.williamspublishing.com
Addison-Wesley
Pearson Education

